INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Recherche multiple d'association d'expressions régulières flexibles dans les grandes séquences génétiques.

Robin Gras

N° 3240

Septembre 1997

THÈME 3

ISSN 0249-6399

Recherche multiple d'association d'expressions régulières flexibles dans les grandes séquences génétiques.

Robin Gras

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet Repco

Rapport de recherche n° 3240 — Septembre 1997 — 52 pages

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 02 99 84 71 00 – Télécopie : (33) 02 99 84 71 71

Résumé : Nous nous intéressons à la recherche de motifs particuliers dans les grandes séquences génétiques. Bien que motivés par une application biologique, les résultats présentés restent applicable à n'importe quel type de séquences. Nous présentons un algorithme permettant la recherche de co-occurrence de motifs de type expression régulière avec erreurs. Nous utilisons le langage A défini par Myers comme formalisme de représentation des motifs complexes que nous recherchons. Ce langage permet de définir l'association de deux motifs comme la présence du second à un intervalle de distances donné du premier.

Notre algorithme est basé sur un pré-traitement de la séquence à analyser, en temps linéaire par rapport à sa taille, sous forme d'arbre des suffixes. Nous utilisons tout d'abord l'arbre comme "support" d'un algorithme de programmation dynamique optimisé qui cherche tous les appariements différents entre le premier motif et les mots de la séquence. Dans une deuxième étape, nous utilisons l'arbre pour trouver toutes les occurrences dans la séquence des appariements préalablement identifiés. Ceci s'effectue en temps linéaire sur le nombre d'occurrences. Enfin, nous utilisons l'arbre pour élaguer les recherches successives des motifs associés en limitant les parcours de l'arbre lors de la première étape aux seuls sous-mots de l'arbre susceptibles de se trouver dans l'intervalle de distance autorisé du motif précédent.

L'ordre dans lequel sont recherchés les sous-motifs influe considérablement sur l'efficacité de la recherche. En effet, moins un sous-motif a d'occurrences et moins il y aura de positions possible pour le prochain sous-motif corrélé et donc plus l'élagage sera efficace. Nous proposons un critère, uniquement basé sur le nombre de sous-mots différents qui peuvent s'apparier avec un sous-motif, représentant le nombre potentiel d'occurrences d'un sous-motif donné par rapport à une séquence aléatoire et cela sans avoir recours à une simulation. Nous utilisons ce critère pour déterminer un ordre de recherche des sous-motifs optimum vis à vis du nombre d'opérations nécessaire pour la recherche du motif complet.

Une fois l'arbre des suffixes construit, cet algorithme permet de faire des recherches de motifs complexes en un temps indépendant de la taille de la séquence. Il s'avère donc particulièrement adapté et meilleur que celui de Myers pour faire des recherches multiples de tels motifs dans les grandes séquences génétiques.

Mots-clé : recherche de motifs, séquences génétiques, arbre des suffixes

(Abstract: pto)

Search for multiple correlated flexible networks expressions in large genetic sequences.

Abstract: We investigate a particular instance of the pattern matching problem. The paper studies an algorithm for (multiple) search in a (large) sequence of patterns belonging to language A developed by Myers [MM93]. This language allows complex regular patterns made of sub-patterns with correlated positions in the sequence, i.e. corresponding to occurrences separated from each other by gaps of variable length. This problem originates from observed constraints in biological macromolecules (DNA, RNA) and is particularly suitable in this domain.

We propose an off-line algorithm, based on a preprocessing of the analyzed sequence, building a suffix tree. We use it in a first step to optimize the dynamic programming algorithm which searches all the different matches between the first sub-pattern and all words of the sequence. In a second step, we use the suffix tree to find all occurrences in the sequence of previously identified matches. This can be done in linear time with respect to the number of occurrences.
Finally, we use the suffix tree to prune subsequent search of coupled sub-patterns in order to restrict the tree parsing. At each step of the algorithm, this allows, before starting the search of the next sub-pattern, to retain only subwords of the tree which could be within the allowed gap of the previous sub-pattern.

The order in which sub-patterns are searched can dramatically influence the efficiency of the search. Indeed, the less is the number of occurrences of a sub-pattern, the less is the number of possible positions for the next correlated sub-pattern and the more the pruning is efficient. We propose a criterion, solely based on the number of different words that can match the pattern, reflecting the potential number of occurrences of a given sub-pattern over a random sequence but that does not requires simulations. We use this criterion to determine an optimal search ordering of sub-patterns with regard to the total number of operations needed to search the whole pattern.

Apart from the preprocessing stage which is linear with respect to the length of the sequence, this algorithm allows to search for complex patterns in time independent of the size of the sequence. It is thus particularly useful when we need to make numerous search of such patterns, as it is the case in an experimental domain like molecular biology. In this respect, we improve Myer's algorithm when we have numerous search of complex pattern in a same sequence.

Key-words: pattern matching, genetic sequences, suffix tree

Table des matières

1 Introduction

Nous nous intéressons à un problème particulier de recherche de motifs (pattern matching) dans une (grande) séquence. Nous étudions dans ce rapport la recherche de motifs appartenant au *langage A* développé par Myers [MM93]. Il s'agit de motifs complexes composés de sous-motifs associés par leurs positions relatives dans la séquence. Ce problème est issu des contraintes observées dans les macro-molécules biologiques (ADN, ARN) et nous décrivons brièvement les spécificités de ce domaine.

Un grand nombre de mécanismes biologiques intervenant sur les macro-molécules fondamentales d'ADN ou d'ARN mettent en jeu la reconnaissance simultanée ou successive de motifs cibles associés. Chaque motif correspond à un site actif où d'autres molécules peuvent venir se fixer et coordonner leurs efforts pour produire une fonction biologique particulière. Plus formellement, deux caractéristiques principales sont importantes dans le processus de reconnaissance des motifs :

La première est due à la nature hiérarchique de la reconnaissance globale. Un "motif" est un ensemble de sous-motifs agissant conjointement ou au contraire inhibant leurs effets entre eux. Chaque sous-motif doit se trouver à une distance relativement stable des autres de façon à ce que les interactions puissent avoir lieu.

La deuxième caractéristique est due à la nature chimique des appariements biologiques. Les appariements approchés sont non seulement fréquents mais la plupart du temps ils constituent même la règle. En effet, la majeure partie des réactions devant être réversibles, les appariements trop forts ne doivent pas se produire. D'un autre côté, il faut malgré tout que les appariements aient lieu. Le mécanisme doit donc être suffisamment "souple" pour accepter des réactions entre molécules ne s'appariant pas parfaitement.

De façon à prendre en compte ces caractéristiques, nous développons un outil de recherche multiples d'expressions régulières flexibles séparées par des chaînes quelconques de longueur bornée (gaps). Nous considérons plus particulièrement dans ce papier, la recherche de "network expression" P (expression régulière sans la fermeture de Kleene) de longueur m, en permettant un nombre limité d'erreurs k, dans une chaîne S de longueur n sur un alphabet Σ. P est composé d'un ensemble de mots (sous-motifs) reliés par des contraintes de séquentialité.

1.1 Définitions et notations

Nous présentons ici de manière plus formelle les définitions et les notations qui vous nous être utiles tout au long de l'article.

Soit Σ un alphabet fini, $\Sigma' = 2^{\Sigma}$ l'ensemble des parties de cet alphabet et ϵ la chaîne vide. Soit $S \in \Sigma^*$ le texte à analyser et $P \in \Sigma'^*$ un sous-motif à rechercher dans S. Si $|S| = n$ et $|P| = m$, on note $S = S_1 \dots S_n$ et $P = P_1 \dots P_m$. $S_{j \dots j'}$ est le sous-mot de S $S_j S_{j+1} \dots S_{j'}$. Une occurrence en j d'un sous-mot w est telle que $w = S_{j-|w|+1} \dots S_j$, j étant la position finale de l'occurrence. On note Nb(w) le nombre d'occurrences de w dans S, c'est à dire le nombre de positions j de S tel que j soit la position finale d'une occurrence de w dans S. Soit $w = v_1 v_2$ tel que $w \in \Sigma^*$, $v_1 \in \Sigma^*$ et $v_2 \in \Sigma^*$, on dit alors que v_1 est un préfixe de w et v_2 est un suffixe de w.

Définition 1 : v est un co-suffixe de w si w est un suffixe de v.

On appelle opération d'édition sur un mot w une substitution, insertion ou suppression d'un caractère de w. On associe une valeur d'une fonction de coût d'édition d à chaque opération d'édition. Si $a \in \Sigma'$ et $b \in \Sigma$, on note $d(a, b)$ le coût d'édition d'une substitution avec $d(a, b) = 0$ si $b \in a$, $d(a, \epsilon)$ le coût d'une suppression et $d(\epsilon, b)$ le coût d'une insertion. La distance d'édition entre deux mots v et w se définit alors par $d(v, w)$, qui est le minimum de la somme des coûts d'édition de toutes séries d'opérations transformant v en w. On peut alors définir un appariement approché avec k_{max} erreurs comme un coût d'édition maximum autorisé entre deux mots.

Définition 2 : Si pour un sous-mot $S_{j' \dots j}$ de S on a $d(P, S_{j' \dots j}) \leq k_{max}$ alors on dit qu'il y a un appariement approximatif de P avec k erreurs à la position j de S.

Dans notre application, n est très grand ($> 10^6$) et le nombre de recherches différentes successives sur une même séquence peut être important du fait de l'aspect expérimental de l'étude des motifs biologiques. De plus, nous voulons être capables de représenter efficacement l'ensemble des solutions obtenues, c'est à dire, l'ensemble des occurrences des mots de S s'appariant avec P.

1.2 Plan du rapport

Dans la section suivante, nous comparons différents algorithmes existant permettant la recherche de motifs complexes tels que nous les avons définis. Nous présentons plus particulièrement l'algorithme de Myers [Mye96] du fait de la grande similitude des problèmes qu'il permet de traiter avec ce que nous proposons.

Nous développons dans la section 3 notre algorithme, linéaire en fonction de la longueur des motifs cherchés et de leur nombre d'occurrences, basé sur le prétraitement de la séquence sous forme d'arbre des suffixes. Notre algorithme tire parti des algorithmes de Chen et Seiferas [CS85] et de Cobbs [Cob95b][Cob95a] pour la première étape permettant la construction de l'arbre des suffixes de la séquence étudiée et la recherche des sous-motifs. Dans une deuxième étape, nous organisons la recherche des sous-motifs avec des contraintes de séquentialité. La représentation finale des occurrences de P se fait sous forme de vecteurs booléens de positions dans la séquence S.

2 Etat de l'art de la recherche de motifs dans des textes

Dans la première partie de cette section, nous présentons un rapide aperçu des différents algorithmes de recherche de motifs dans un texte en les classant suivant deux dimensions liées au type de motifs cherchés ainsi qu'à la structure de la recherche. Dans la seconde, nous montrons la nécessité d'un algorithme spécifique pour la recherche de motifs corrélés dans les séquences biologiques. Enfin, dans la dernière partie, nous présentons plus en détail l'algorithme de Myers [Mye96] permettant une recherche de motifs tels que nous les avons définis précédemment.

2.1 Algorithmes généraux de recherche de motifs

Un grand nombre d'algorithmes ont été développés pour la recherche de motifs dans les textes [EM96][BJEG95]. Ces algorithmes peuvent être classifiés suivant plusieurs dimensions, leur donnant à chacun un champ d'application privilégié. A travers un rapide bilan de la littérature, nous allons montrer les forces et faiblesses des principaux algorithmes existant vis à vis des contraintes que nous avons définies.

Nous avons choisi deux dimensions principales pour caractériser les différents types d'algorithmes.

La première concerne la nature des motifs recherchés, conduisant à deux types de recherches : la *recherche exacte* et *la recherche approchée*.

Dans le premier cas, on veut trouver la position dans un texte S de toutes les occurrences exactes d'un motif P donné. Dans le deuxième, on veut trouver la position dans S de tous les mots ressemblants à P, c'est à dire ne différant de P que sur un petit nombre de caractères. On peut définir le degré de ressemblance par la distance d'édition, qui correspond au nombre minimal de substitutions, d'insertions et de suppressions de caractères à effectuer sur un sous-mot de S pour être identique à P. Si l'on note k ce nombre d'opérations, le premier cas correspond à $k = 0$ et le deuxième est un appariement approché avec k erreurs.

On peut autoriser aussi une ambiguïté sur la nature d'un caractère à une position donnée. Un caractère ambigu A est alors tel que $A \in 2^{\Sigma}$, c'est à dire que l'on accepte à une position du motif un appariement avec un sous-ensemble de caractères de Σ à la position correspondante du texte. Une erreur sur A est alors un caractère de Σ/A. Par exemple, avec $\Sigma = \{a, b, c\}$ et $A = \{a, b\}$, le mot Aa s'apparie avec le mot ba sans erreur et avec le mot ca avec une erreur.

La deuxième dimension est liée au mode de recherche des motifs dans le texte et l'on peut aussi distinguer deux cas. La *recherche "on line"* concerne les algorithmes qui analysent directement le texte. Ils ont donc l'avantage de ne nécessiter que la gestion de structures de données peu volumineuses par rapport à la taille du texte. La *recherche "off line"* est basée sur un pré-traitement du texte à analyser. Cela conduit en général à la construction d'un index sur le texte (arbre ou automate des suffixes) qui peut augmenter fortement la taille de sa représentation. Le but est dans ce cas d'optimiser les recherches ultérieures de motifs grâce à cet index.

Les premiers algorithmes développés ont été des algorithmes de recherche exacte. Boyer-Moore [BM77] et Knuth-Morris-Pratt [KMP77] ont mis au point les algorithmes les plus efficaces dans le cas de la recherche on line. Ils consistent en un déplacement le long du texte d'une fenêtre de la longueur du motif et comparent les caractères de la fenêtre avec ceux du motif. Leur efficacité provient du fait qu'à chaque étape de l'algorithme, on prend en compte l'information obtenue à l'étape précédente pour minimiser le nombre de caractères à comparer. Le premier a ainsi une complexité en $O(m+n)$ et le deuxième en $O(n+rm)$ avec r le nombre d'occurrences de P dans S[1].

1. Dans le cas où l'alphabet Σ est grand sa complexité en pratique se rapproche de $O(n/m)$

La recherche off line [AP83][Apo85] est basée sur la construction d'un arbre des suffixes [Wei73] [McC76] [CS85] [Ukk93] ou d'un automate des suffixes [BBH$^+$85] [Cro88]. Le temps de construction de ces structures de données ainsi que l'espace nécessaire à leur construction est linéaire par rapport à la taille du texte. L'intérêt de l'utilisation de ces structures est que la recherche de motifs exacts dans le texte après leur construction ne dépend plus de la longueur du texte mais uniquement de la longueur des motifs cherchés. On les utilise comme un dictionnaire des mots contenus dans le texte en parcourant l'arbre ou l'automate sur autant d'états qu'il y a de caractères dans le motif. Ils permettent ensuite d'accéder à la position dans le texte de toutes les occurrences du motif en un temps linéaire en fonction de ce nombre d'occurrences.

Bien que ces algorithmes soient très utiles pour trouver l'ensemble des répétitions exactes d'un motif dans le texte [GN96], dans de très nombreux problèmes la recherche sans erreurs ne suffit pas. Ainsi, lors de la recherche d'un mot dans un texte, un tel algorithme de recherche permet de retrouver tous les mots ayant une orthographe approchante. De même, un grand nombre de fonctions biologiques intervenant sur les séquences génétiques (transcription de gènes, structure tridimensionnelle des molécules) impliquent des appariements de molécules imparfaits. Il est donc nécessaire de rechercher ces zones d'appariements en tenant compte du caractère flexible des motifs impliqués.

Une seconde classe d'algorithmes à été développée pour répondre à ces problèmes, basée sur la recherche de motifs approchés ($k > 0$) dans un texte. De la même façon que dans le cas de la recherche exacte, on peut distinguer deux classes d'algorithmes : avec ou sans pré-traitement du texte.

Dans le cas des algorithmes on line, la programmation dynamique a été fréquemment utilisée dans les premiers algorithmes, conduisant à des temps de recherche en $O(mn)$. Puis, des algorithmes basés sur l'approche Boyer-Moore ou Knuth-Morris-Pratt [LV88a] [TU93] ou sur une optimisation des algorithmes de programmation dynamique [LV88b] [GP90] [Ukk85] ont proposé des recherches en $O(kn)$. D'autres, suivant une approche numérique basée sur la compilation du motif cherché sous forme numérique et tirant parti de la capacité des langages de programmation à manipuler les mots machines (algorithme shift-add [BYG92]), effectuent la recherche en $O(n)$ quand la taille du motif est de l'ordre de celle du mot machine. Plus récemment, un algorithme tirant parti à la fois des propriétés de l'approche Boyer-Moore et des algorithmes numériques [EMC96], a conduit à un temps de recherche en $O(n + \frac{k+4}{m-k})$ dans le cas ou l'alphabet est suffisamment grand par rapport à la taille

du motif. Dans le cas des petits alphabets (alphabet de séquences de nucléotides par exemple), pour que l'algorithme reste efficace, il faut que la longueur du motif soit très supérieur au nombre d'erreur. Il est ainsi particulièrement efficace dans le cas de recherche de motif longs car il permet un temps de recherche décroissant avec la taille du motif. Toutefois, la longueur du motif ne doit pas être trop supérieur à la taille d'un mot machine pour conserver son efficacité. Cet algorithme prend aussi en compte la possibilité d'ambiguïté sur les caractères du motif lors de sa compilation.

Dans le cas off line, les algorithmes sont basés, comme pour la recherche sans erreurs, sur un pré-traitement du texte en $O(n)$ de façon à construire un index permettant d'accélérer les recherches ultérieures des motifs. Un premier type de pré-traitement est basé sur le codage du texte à l'aide des mots de longueur $L = \log_{|\Sigma|} n$ le composant. On crée ainsi un nouvel alphabet Σ' de taille $|\Sigma|^L$ pouvant servir pour représenter le texte. Karlin [KMGL87] utilise cette méthode pour réécrire le texte sur ce nouvel alphabet, obtenant un nouveau texte S' de longueur $n - L + 1$. Il utilise ensuite S' pour faire ses tests d'appariements exacts de mots de longueur L en $O(1)$. Myers [Mye94] utilise le même type de codage mais cette fois pour produire un index des mots de longueur L du texte. Il crée $|\Sigma'|$ listes contenant les positions des occurrences dans S des mots de longueur L. Il construit ensuite la liste des appariements entre P et S récursivement à partir des appariements entre P et les mots de longueur l, l étant multiplié par deux à chaque étape et variant entre L et m. Il arrive ainsi à un algorithme en $O(kn^{pow(\epsilon)} \log n)$ avec $\epsilon = k/m$ et $pow(\epsilon) < 1$ pour k suffisamment petit par rapport à m. Ces algorithmes étant fortement liés à la taille de l'alphabet sont beaucoup plus efficaces dans le cas d'un petit alphabet.

Comme dans le cas de la recherche exacte, le deuxième type de pré-traitement utilisé est basé sur la construction d'un arbre des suffixes. Ukkonen [Ukk93] propose plusieurs algorithmes utilisant cette structure avec un temps de recherche au mieux en $O(mq \log q + nO)$ avec $q = O(min(n, m^{k+1}|\Sigma|^k))$ correspondant au nombre de préfixes de l'arbre des suffixes examinés et nO le nombre d'occurrences du motif. L'idée est d'utiliser l'arbre des suffixes pour optimiser l'algorithme de programmation dynamique de recherche de motifs avec erreurs. Il constate que dans la matrice de programmation dynamique, un grand nombre d'entrées sont redondantes et conduisent donc à du calcul inutile. Il construit par étapes en fonction de m l'ensemble des sous-mots différents de S viables, c'est à dire ceux pour qui la distance d'édition avec P est inférieure à k, cet ensemble étant au plus de taille q. Il obtient, à la dernière étape, la liste des sous-mots différents de S s'appariant avec moins de k erreurs avec P. Il utilise ensuite de nouveau l'arbre des suffixes pour trouver leurs positions dans S en un temps proportionnel à leur nombre d'occurrences. Cobbs

[Cob95b] reprend le principe de cet algorithme en optimisant l'élagage des sous-mots viables en ne gardant que ceux qui ne sont pas co-suffixes d'un autre sous-mot viable ayant une distance d'édition au moins aussi faible que la leur. La complexité de son algorithme est ainsi en $O(mq + nO)$. Etant donné que nous utilisons cet algorithme lors de notre recherche de motifs complexes, nous le décrivons plus en détail dans la section 3.

2.2 Algorithme de recherche de motifs corrélés dans les séquences biologiques

2.2.1 La recherche de motifs corrélés

Les motifs complexes que nous voulons trouver dans un texte sont des expressions régulières "limitées" c'est à dire n'acceptant que la concaténation et l'union (sans les répétitions illimitées) nous permettant de représenter dans un même motif l'union de plusieurs motifs contenant des caractères ambigus. Nous voulons faire une recherche approchée de ces motifs et donc autoriser un nombre k d'erreurs lors de l'appariement avec un sous-mot du texte. De plus, nous nous intéressons à l'association de tels motifs. Nous voulons donc pouvoir spécifier un intervalle définissant la distance acceptée entre leurs différentes occurrences ou la distance "d'exclusion" c'està dire la distance à laquelle on ne veut pas trouver leur présence commune. Cet intervalle de distance doit pouvoir prendre des valeurs négatives de façon à pouvoir rechercher la présence d'un motif "autour" d'un autre. Enfin, nous voulons faire des recherches multiples de ce type de motif sur un même texte, nous voulons donc que les temps de recherche individuels ne dépendent pas de la taille du texte.

Aucun des algorithmes présentés précédemment ne répond entièrement à ces critères. Bien évidemment, les algorithmes spécifiques de recherche exacte ne peuvent pas convenir. Dans le cas des algorithmes de recherche approchée, les algorithmes on line nécessitent par définition un temps dépendant de la taille du texte pour chaque recherche (même si l'algorithme de Myers [Mye94] est sous-linéaire en fonction de la taille du texte, le facteur est relativement faible: de l'ordre de 1/3 dans le cas d'un petit alphabet (4)). De plus, les algorithmes basés sur la compilation des motifs sous forme numérique sont limités dans la taille des motifs analysés par la taille des mots mémoires représentés en machine. Ils ne permettent pas non plus de prendre en compte les erreurs de type insertions et suppressions.

2.2.2 Prise en compte des intervalles de distance entre motifs

Ce qui nous intéresse plus particulièrement dans les spécifications que nous avons données des motifs que nous cherchons est la possibilité d'étudier l'association de motifs. Aucun de ces algorithmes ne permet de prendre en compte des intervalles de distances entre motifs.

Une approche naïve serait de concevoir l'association de deux motifs u et v comme un seul motif w tel que $w = uGv$ avec G une série de longueur g variable acceptant n'importe quel caractère de Σ. On peut le voir comme la recherche du motif uGv avec $k = k1 + g + k2$ erreurs, k1 erreurs sur u, g entre u et v et k2 erreurs sur v. Cela pose malheureusement plusieurs problèmes :

- Il faut pouvoir spécifier la position des erreurs permises dans l'intervalle entre u et v tout en laissant les positions des k1 et k2 erreurs variables dans u et v.
- L'intervalle entre u et v fait partie du motif cherché, augmentant fortement sa taille.
- Le nombre d'erreurs total accepté dans le motif devient très important.

Pour toutes ces raisons, il est important de trouver une meilleure représentation des intervalles entre motifs. Les seuls algorithmes à notre connaissance permettant la recherche de motifs associés avec espacements tels que les motifs définis dans la banque de protéines PROSITE [Bai92] sont les algorithmes de Jonassen [JCH95] [JE95] et de Myers [Mye96]. Jonassen présente un outil, Pratt, permettant de rechercher dans un ensemble de séquences protéiques le "meilleur" motif présent dans toutes les séquences. Son algorithme est basé sur la recherche des blocs de tailles réduites conservés dans toutes les séquences puis par leurs extensions, en autorisant des espacements de tailles variables entre les différentes parties du motif. Cet algorithme ne correspond toutefois pas au problème que nous voulons traiter car il s'intéresse à un problème combinatoirement beaucoup plus complexe : la recherche automatique de tous les motifs communs à un ensemble de séquences. Myers, par contre, présente un algorithme répondant parfaitement au problème que nous avons posé. Nous présentons donc dans la partie suivante le formalisme proposé par Myers [MM93], permettant une représentation simple de la notion d'intervalle et présentant un algorithme efficace de recherche de motifs associés.

2.3 Algorithme on line de recherche de motifs avec erreurs de Myers

Myers [Mye96] décrit un algorithme permettant une recherche on line de motifs en réseau (network patterns) avec erreurs, séparés par des intervalles (spacers). Cet algorithme utilise une représentation des networks patterns (sous-motifs) sous forme d'automate fini non déterministe. Il défini un ordonnancement de la recherche des sous-motifs de façon à minimiser le temps de calcul, puis à partir des occurrences du premier sous-motif cherché, il étend sa recherche aux sous-motifs suivants sous la contrainte de distance imposée par les intervalles.

2.3.1 Le Langage A

Nous présentons informellement le langage A de représentation de motifs choisi par Myers [MM93]. Celui-ci permet la représentation des motifs biologiques classiques tels que ceux trouvés dans les promoteurs [MHEM84], ainsi que celle de motifs complexes plus récents tel que la Cytosine Methyltransferase [PBPR89].

Un sous-motif se représente sous la forme d'un network pattern.

Motif A = "$ac(gg|ta)ct(gt)?$"

permet par exemple de représenter les quatre mots *acggct, acggctgt, actact et actactgt* (le? signifiant la présence de 0 ou 1 fois l'expression précédente).

Un motif peut alors s'écrire ainsi :

$$\{A,1\}(<0,20>\{B,1\}|<-5,5>\{C,1\})<25>\{D,1\}$$

où A, B, C et D sont des sous-motifs, $\{A,1\}$ signifie que l'on accepte 1 erreur dans le motif A et $\{A,1\}<0,20>\{B,1\}$ que l'on recherche le motif B entre 0 et 20 caractères après le motif A. Les valeurs négatives dans les intervalles signifient que l'on cherche la présence du début du deuxième motif avant la fin du premier.

Le nombre d'erreurs peut aussi se représenter sous la forme d'un taux de reconnaissance, c'est à dire le pourcentage des caractères du motif qui s'apparient sans erreur avec la séquence. Le caractère "." dans un sous-motif représente l'union de tous les caractères de l'alphabet, donc un caractère qui s'apparie sans erreur avec tout caractère de l'alphabet.

Afin d'illustrer la complexité possible des motifs utilisés en biologie, nous donnons celui de la Cytosine Méthyltransférase écrit dans le langage A :

motif I = "[ILM][DS][FL]F[ACS]G.[GM][AG][FIL]..[AGS]....G";
motif II = "[ILV]..[INS][DE].[DFN]..[AI]..[STV][FIY]..[IN]";
motif III = "D[IV][RST]";
motif IV = "[DN].[ILV].[AGS]G[FPS]PC[PQ].[FW]S..G.....[EDS]";
motif V = "[EDP].[QR][GN].[LMV][FY]";
motif VI = "[PT].....ENV.[GN].....[GKN]";
motif VII = "[DG]Y.[FIV]";
motif VIII = "[DIN][ADS]..[FHY][FGN][ILV][AP]Q.R[EKQ]R...[EIV][ACG]";
motif IX = "R.[FLM][HTS]..E..[ARV][ILV][MQ].[FY][DEP]";
motif X = "[KRS]....Y[KQR][EMQ].GN[AS][IV].[IPV].[ALV]....[AFG]";
net MTaset = $\{I,t\} < -9,39 > \{II,t\} < -5,20 > \{III,t\} < -4,34 > \{IV,t\} < -13,41 > \{V,t\} < -1,19 > \{VI,t\} < 1,42 > \{VII,t\} < -7,21 > \{VIII,t\} < 34,322 > \{IX,t\} < -5,25 > \{X,t\}$;

2.3.2 Algorithme de recherche de motif à l'aide d'automate fini déterministe

On va construire un automate fini déterministe (DFA) $A =< Q, \delta, \Sigma_q, \Sigma_\delta, I, F >$ avec Q est un ensemble d'états $q_i(ai)$ où $ai \in \Sigma \cup \{\epsilon\}$, δ est une fonction de transition de $Q \times \Sigma_\delta \rightarrow Q$, Σ_q est l'alphabet sur lequel est construit la séquence, Σ_δ est l'ensemble $\{ins(a),\ sup(a),\ sub(a,b) \mid a,b \in \Sigma_q\}$, I est un état initial et F est un état final.

Toute "expression en réseau" (ER)[2] R peut être transcrite en un automate A (dont la structure est celle d'un $\epsilon - NFA$ reconnaissant le langage décrit par l'expression) dont nous détaillons ensuite la construction. Pour rechercher les occurrences de R dans un texte S de longueur n, il faut explorer un espace de recherche que l'on peut représenter par un *graphe d'alignement* C entre R et S. Ce graphe correspond aux possibilités de progression avec erreurs selon les 2 dimensions que représentent le texte et l'expression recherchée. Il consiste en $n+1$ copies de A, les unes au dessus des autres, une pour chaque caractère du texte (voir figure 1). Les erreurs sont représentées par des arcs de type suppressions auxquels on associe un poids $d(S_i, \epsilon)$, insertions de poids $d(\epsilon, q_s)$ et substitution de poids $d(S_i, q_s)$. L'idée est de parcourir

2. traduction de "network expression" utilisé par Myers

ce graphe par un algorithme de programmation dynamique. En parcourant le graphe à partir du noeud $C(0, I)$ (correspondant au début du texte associé à l'état initial de A) jusqu'au noeud $C(n, F)$ (correspondant au dernier caractère de S associé à l'état final de A), on peut trouver toutes les occurrences de R dans S en un temps en $O(np)$.

A tout moment, l'algorithme n'a besoin que de 2 caractères du texte. La représentation du graphe peut donc être considérablement réduite en observant que à tout moment seules les DFA C_i et C_{i-1} (correspondant aux lignes i et i-1 de C) sont nécessaire au calcul du chemin dans le graphe. Il suffit donc de conserver deux représentations de A, reliées par les arcs insertions, suppressions et substitutions, pour réaliser la recherche sur tout le texte.

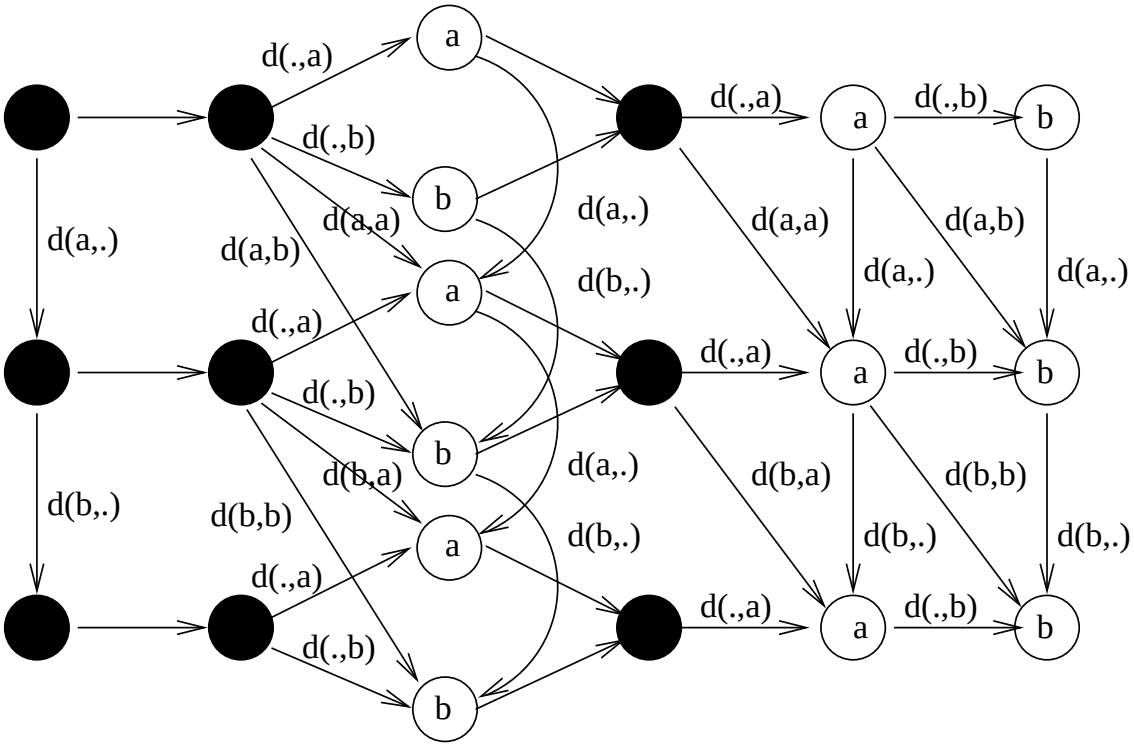
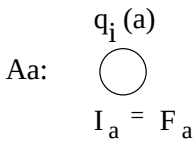


FIG. 1 – *Graphe d'alignement entre $S = ab$ et $R = (a|b)ab$ (le . remplace le ϵ)*

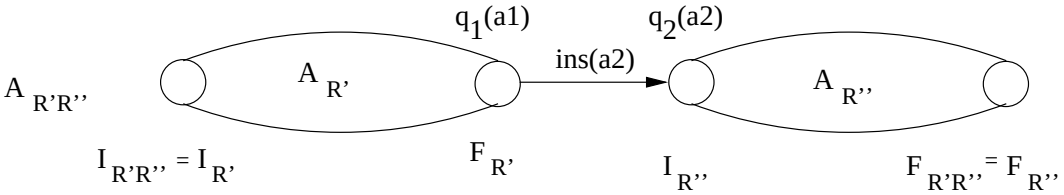
Nous détaillons maintenant la construction en deux étapes de cet automate nécessaire pour réaliser l'alignement entre R et S.

Dans la première étape on construit l'automate A représentant l'expression en réseau R. Les arcs construits a cette étape correspondent aux insertions de caractères du graphe d'alignement. La construction se fait grâce à quatre règles de composition d'expressions élémentaires :

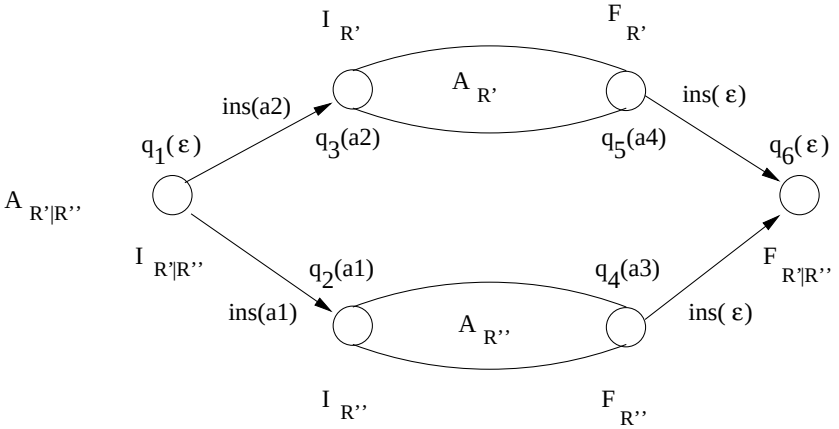
– Si R est réduit à un caractère unique a, le DFA correspondant est tel que :



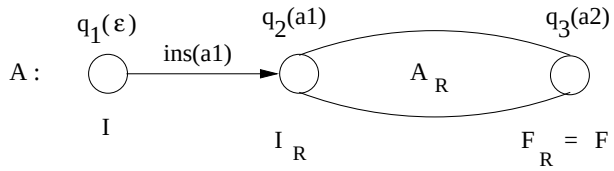
– Si R est la concaténation de deux sous ER R' et R", $A_{R'R''}$ est tel que :



– Si R est l'union de deux sous ER $R'|R''$, $A_{R'|R''}$ est tel que :



– L'état initial du DFA A reconnaissant R est tel que :



Du fait de cette méthode de construction, on s'aperçoit qu'à chaque étape on ajoute au plus deux états dans le DFA et donc on a : $|Q| \leq 2|R|$. Du fait de l'absence de la fermeture de Kleene dans les ER, chaque règle de construction ne permet de créer que des transitions orientées dans le même sens. Les DFA ainsi produits sont donc toujours des DAG (Directed Acyclic Graphs).

Une deuxième étape permet de connecter deux copies de l'automate, A et A', par des arcs correspondant aux suppressions et substitutions du graphe d'alignement. On ajoute un arc suppression entre chaque état de A et son correspondant dans A' et un arc substitution entre chaque état de A et les états de A' correspondant a ses successeurs dans A (voir figure 2).

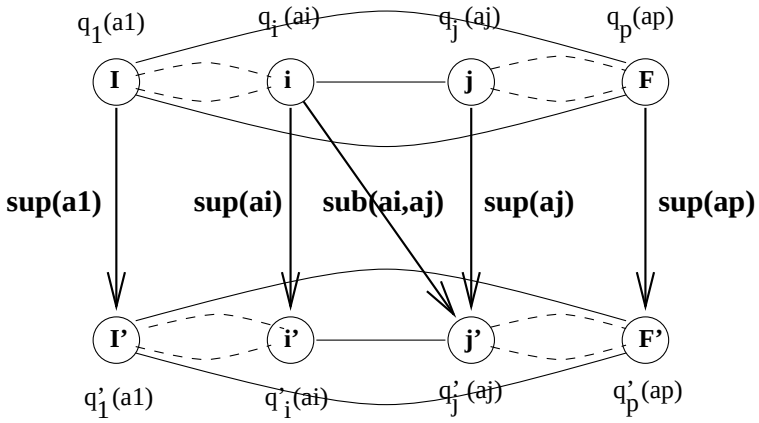


FIG. 2 – *Connexion entre les deux copies du DFA A*

On peut ainsi déterminer que tout noeud (i, q_s) du graphe ou de l'automate final a au plus 5 arcs entrants:

– si $i > 0$, il existe un arc suppression à partir du noeud $(i-1, q_s)$ de poids $d(S_i, \epsilon)$.

- si $q_s \neq I$, pour tout état q_t tel que $q_t \rightarrow q_s$ il existe un arc insertion à partir du noeud (i, q_t) de poids $d(\epsilon, q_s)$
- si $q_s \neq I$ et $i > 0$, pour tout état q_t tel que $q_t \rightarrow q_s$ il existe un arc substitution à partir du noeud $(i-1, q_t)$ de poids $d(S_i, q_s)$

Par construction du DFA grâce au règles de compositions d'expressions élémentaires, pour tout état s du DFA il existe au plus 2 noeuds q_t tel que $q_t \rightarrow q_s$. Il y a donc bien au plus 5 arcs entrant par noeud du DFA et donc sa taille totale est en $O(p)$ avec p la longueur de R. La taille de la structure de données nécessaire pour réaliser l'alignement entre R et S est alors en $O(n+p)$.

2.3.3 Optimisation de l'algorithme dans le cas d'un seuil d'erreur

On peut remarquer que si on donne un nombre maximum d'erreurs k dans l'appariement entre R et S, toutes les valeurs $C(i, q_s)$ de C ne doivent pas être calculées. Seules les valeurs pour lesquelles l'appariement se fait avec un nombre d'erreurs inférieur au seuil doivent être considérées. Ukkonen [Ukk85] à présenté un algorithme tenant compte de cette propriété qui lui permet d'avoir un temps de recherche des occurrences dans le texte en $O(kn)$. L'idée est de ne conserver, à chaque étape de l'algorithme, qu'une certaine "zone" du graphe nécessaire pour les calculs de l'étape suivante et dans laquelle tous les noeuds ont une valeur inférieure au seuil. A chaque étape i, on calcule les valeurs des noeuds de la ligne C_i du graphe en fonction de celles de C_{i-1} (dans le cas des erreurs de type suppressions et substitutions) et celles de C_i (dans le cas des erreurs de type insertions).

Pour ce faire, Myers propose une modification de l'algorithme de programmation dynamique classique. Il faut pour cela connaître à chaque étape i l'ensemble d'états $T_{i-1} = \{q_s : C_{i-1}(q_s) \leq T\}$ pour lesquels la valeur dans la ligne C_{i-1} est inférieure à T. La difficulté est de construire cette liste en un temps constant pour chacun de ces éléments. En fait, c'est un ensemble, $Z_{i-1} \supseteq T_{i-1}$ appelé zone dont les éléments sont les valeurs $C^*_{i-1}(q_s)$ avec $s \in Z_{i-1}$ telles que $C^*_{i-1}(q_s) = C_{i-1}(q_s)$ si $C_{i-1}(q_s) \leq T$ et $C^*_{i-1}(q_s) > T$ sinon, qui va être calculé. Les contraintes appliquées sur la liste Z_{i-1} pour répondre aux critères de temps de construction sont : (1) que le sous graphe de F représentant les noeuds de Z_{i-1} soit connecté de façon à maintenir un ordonnancement des noeuds de la zone et (2) que le retrait de tout noeud de $Z_{i-1} - T_{i-1}$ déconnecte le graphe de façon à maintenir une cardinalité minimale à la zone.

Les noeuds de C_i de valeur inférieure au seuil sont déterminés à partir des noeuds de C_{i-1} en suivant un lien correspondant à une suppression ou une substitution et de noeuds de C_i en suivant des liens correspondant à des insertions. Le calcul de Z_i se fait alors en deux étapes : On calcul $U_i = Z_{i-1} \cup Inserts(Z_{i-1})$ tel que $Inserts(X) = \{q_s : \exists q_t \in X, q_t \rightarrow q_s ou \exists q_t \in Inserts(X), (q_t \rightarrow q_s et C_i(q_t) \leq T)\}$, puis on retire des noeuds de U_i en respectant les critères de connectivité pour arriver à la liste Z_i.

Le calcul de la valeur d'un noeud q_u de U_i nécessite que la valeur de tout les prédécesseurs de q_u soient connus. Cela implique que les noeuds de U_i soient évalués dans un ordre topologique correspondant à la structure du graphe. Myers introduit alors un typage des noeuds et des arcs du graphe en fonction de la structure de l'automate (un noeud est élément d'une partie de l'automate de la forme F_{RS} ou $F_{R|S}$). On peut ainsi connaître à tout moment lors du calcul de la valeur d'un noeud la position où il faut l'insérer dans la liste U_i en fonction de la position de la liste de ces prédécesseurs.

Par cette méthode, il obtient un algorithme de complexité en $O(tn)$ avec t la taille moyenne de la liste Z. Il ne donne toutefois pas la preuve que cette complexité est bien du même ordre que $O(kn)$ mais il fait une comparaison avec l'algorithme d'Ukkonen pour lequel cette démonstration à été faite.

2.3.4 Ordonnancement de la recherche dans le cas de motifs séparés par des intervalles

Myers s'intéresse ensuite à la recherche d'un motif "linéaire" constitué de plusieurs sous-motifs de type expression en réseau séparés par des intervalles de distances dont les bornes peuvent être positives ou négatives. Soit $M = M_0 S_1 M_1 S_2 M_2 \ldots S_p M_p$ un tel motif avec M_i *pour* $0 \leq i \leq p$ les sous-motifs de longueur p_i et S_i *pour* $1 \leq i \leq p$ les intervalles de longueurs Δ_i. Le principe est de rechercher l'un de ces sous-motifs dans le texte entier puis, à partir des positions d'appariement trouvées, de rechercher les sous-motifs suivants. Il remarque que le choix du premier sous-motif est primordiale pour l'efficacité de la suite de la recherche. En effet, moins la premier sous-motif aura d'occurrences dans le texte, moins il y aura de positions à considérer pour la recherche des sous-motifs suivants. De plus, pour tout motif supplémentaire trouvé, le nombre de positions ne peut que décroître. C'est donc l'ordre dans lequel seront recherchés tous les sous-motifs qui est important pour diminuer le plus rapidement possible le nombre de positions à considérer. Il détermine pour cela une estimation par la méthode de Monte Carlo de la fréquence f d'apparition de chacun des sous-motifs dans des séquences aléatoires. Le temps nécessaire pour la recherche

d'un sous-motif M_k par l'algorithme défini précédemment est de l'ordre de $O(t_k\Delta_k)$ avec Δ_k l'intervalle de distance par rapport au motif précédent et t_k fonction du nombre d'erreurs. Sachant la position des sous-motifs M_k à M_h, il donne une équation de récurrence permettant de déterminer le meilleur choix pour le sous-motif suivant :

$$Best(k+1,h-1) = min(\Delta_{k+1} f_k Best(k,h-1) + t_k\Delta_{k+1}, \Delta_h f_h Best(k+1,h) + t_h\Delta_h) \quad l$$
$$et\ Best(0,p) = 0$$

Avec f_k (resp. f_h) la fréquence supposée d'apparition du sous-motif M_k (resp. M_h) dans S. Pour déterminer ensuite le meilleur ordonnancement complet de tous les sous-motifs, il suffit de calculer la valeur de $opt(k) = nt_k + nf_k Best(k,k)$ pour tous les sous-motifs M_k de M. Grâce à un algorithme de programmation dynamique il peut déterminer la meilleure valeur de $opt(k)$ en $O(p^2)$ avec p la longueur totale du motif M et ainsi trouver l'ordonnancement optimal de la recherche des sous-motifs.

2.3.5 Recherche des appariements et analyse de complexité

Une fois déterminé l'ordre sur [-p,q] dans lequel doivent être recherchés les sous-motifs, il faut réaliser de manière efficace la recherche d'un motif de la forme : $M_{-p}S_{-p}M_{-(p-1)} \ldots M_{-1}S_{-1}M_0S_1M_1 \ldots M_{q-1}S_qM_q$. Myers défini pour cela deux fonctions. La fonction Scan(M,J) qui effectue la recherche du sous-motif M dans un sous ensemble de positions J de S. Cette fonction correspond à la recherche avec un seuil d'erreur et fournit en résultat un intervalle de positions finales des appariements de M aux positions J en un temps $O(t|J|)$. La fonction $Space_k(J) = \cup_{j\in J}[j+l_k, j+r_k]$ pour l'intervalle $S_k = [l_k, r_k]$ permettant d'étendre un intervalle à droite d'une liste de positions.

L'ordonnancement des motifs implique que la recherche du sous-motif suivant ne se fait pas toujours dans la même direction : il peut se faire à gauche ou à droite du sous-motif courant. Or les fonctions précédentes sont orientées pour une recherche à gauche. Myers propose donc des fonctions inverses : $Scan^r$ et $Space^r_k$. $Scan^r(M,J)$ nécessite la construction de l'automate inversé de M et permet par un parcours de droite à gauche de S de trouver l'ensemble des positions de début d'appariement entre M et J. $Space^r_k = \cup_{j\in J}[j-l_k, j-r_k]$ permet d'étendre l'intervalle S_k à droite de la liste de position J.

A l'aide de ces quatre fonctions, Myers propose un algorithme permettant la recherche d'un motif linéaire de la forme présenté plus haut.

algorithme 1 algorithme de Myers

pour tout $[a,b] \in Scan(M_0,[0,n])$ **faire**
  {recherche des positions droites du premier motif}
  $X \leftarrow L_0 \leftarrow Scan^r(M_0,[a,b])$ {recherche des positions gauche correspondantes et création de la liste des intervalles de positions du premier motif}
  pour tout k *dans l'ordonnancement sur* $[-p,-1] \cup [1,q]$) **faire**
    si $X = \emptyset$ **alors**
      break
    fin si
    si $k < 0$ **alors**
      $X \leftarrow L_{-k} \leftarrow Scan^r(M_k, Space_k^r(L_{-k-1}))$
    sinon
      $X \leftarrow R_{-k} \leftarrow Scan(M_k, Space_k(R_{k-1}))$
    fin si
  fin pour
fin pour

A partir de cet algorithme on peut déterminer facilement la complexité de la recherche d'un motif complet. La première boucle correspond à la recherche dans le texte entier du premier sous-motif et à donc une complexité en $O(tn)$ avec t fonction du nombre d'erreurs autorisées sur ce sous-motif. Si l'on note Q le nombre de positions d'appariement du premier sous-motif dans S, on peut estimer la complexité de la recherche des sous-motifs suivants. Pour toutes les positions Q du premier sous-motif et pour tous les m sous-motifs suivants on a un temps de recherche proportionnel à t fois la taille de sous-motif cherché. Si on note Δ la taille moyenne d'un intervalle et p la taille moyenne d'un sous-motif, la complexité totale de l'algorithme est en $O(tn + Qmt(\Delta + p)$.

Myers à développé le programme ANREP [MM93] basé sur cet algorithme. Il a comparé le temps d'exécution de son programme par rapport à un programme basé sur un algorithme de programmation dynamique classique. Il a pu vérifier ainsi la sensibilité de son programme au nombre d'erreurs autorisé : moins il y a d'erreurs et plus ANREP est rapide et souvent meilleur que l'algorithme classique. Il a déterminé à partir de ces expérimentations un seuil sur la taille de la zone Z (égal au paramètre t intervenant dans la complexité) par rapport à la taille du NFA pour que ANREP soit plus rapide que l'algorithme classique. Le seuil se situe à une taille de Z inférieure à 1/3 de la taille du NFA. Avant de lancer la recherche d'un motif particulier, il utilise une simulation sur une séquence aléatoire courte de la recherche de ce motif

pour déterminer si la taille moyenne de la zone Z est inférieure à 1/3 de la taille du NFA et donc choisir entre l'utilisation de l'algorithme optimisé ou de l'algorithme classique.

3 Une nouvelle approche pour la recherche de motifs complexes

Nous présentons dans cette section notre approche de la recherche de motifs complexes. Nous donnons tout d'abord notre algorithme général de façon à avoir une mesure précise de sa complexité qui est nécessaire pour le calcul de l'ordonnancement. Nous développons ensuite les différentes parties de l'algorithme dans l'ordre de leur utilisation.

3.1 L'algorithme général

Notre algorithme peut se décomposer en trois parties. Premièrement, nous reprenons la méthode de Myers pour déterminer l'ordonnancement de la recherche des sous-motifs de façon à minimiser le plus rapidement possible l'espace de recherche. Nous adaptons toutefois sa méthode pour tenir compte des différentes phases de notre algorithme et nous proposons une extension permettant de remplacer l'estimation de la fréquence d'apparition d'un sous-motif par simulation sur une séquence aléatoire par le calcul direct de cette estimation.

Dans un deuxième temps, nous utilisons l'algorithme de Cobbs, adapté pour prendre en compte l'ambiguïté sur les caractères, pour faire la recherche d'un sous-motif particulier en s'appuyant sur l'arbre des suffixes.

Enfin, après chaque recherche de sous-motif, nous passons par une phase de marquage de l'arbre de façon à repérer l'ensemble des sous-mots susceptibles de conduire à des occurrences du prochain sous-motif se trouvant dans l'intervalle de distance autorisé du sous-motif précédent. Ceci nous permet d'accélérer la phase de recherche par l'algorithme de Cobbs du sous-motif suivant en "élaguant" les branches de l'arbre ne répondant pas aux critères de contiguïté.

La complexité totale de l'algorithme 2, qui suppose l'ordre des suffixes du texte construit est alors en $O(q(m(|\Sigma| + 2k) + (\Delta + 2k)\log_{|\Sigma|} n))$ (que l'on peut estimer à $O(q(m(4 + 2k) + 10(\Delta + 2k)))$ pour des recherches classiques sur des séquences d'ADN de l'ordre du million de caractères) et donc bien indépendant de la taille du texte ($\log_{|\Sigma|} n$ pouvant être considéré comme négligeable par rapport à l'ordre de grandeur des textes qui nous intéressent).

algorithme 2 Procédure recherche_motif(M, arbre des suffixes)

liste_position $\Leftarrow$ *nil* {On met la liste des positions du motif a vide}
ordonnancement(M, O) {calcul du meilleur ordonnancement de M avec m_1 le premier sous-motif}
pour tout $(m_i, \Delta_i) \in O$ **faire**
  recherche_sous_motif(m_i, Δ_i, liste_position, arbre des suffixes)
fin pour{lancement de la recherche}

procédure recherche_sous_motif(m, Δ_i, liste_position, A)
si Union(m) **alors**
  {si le sous-motif m est constitué d'une union}
  liste_union $\Leftarrow$ *nil* {liste de construction des positions de l'union}
  pour tout $m_i \in m$ **faire**
    liste_inter $\Leftarrow$ *liste_position*
    recherche_sous_motif(m_i, Δ_i, liste_inter, A)
    liste_union $\Leftarrow$ *liste_union* + *liste_inter*
  fin pour
  liste_position $\Leftarrow$ *liste_union*
sinon
  Cobbs(m, liste_position, A) {recherche des positions du sous-motif m}
  extension_gauche(m, liste_position) {extension à gauche des débuts de positions de m}
  remise_à_zéro(A) {remise à zéro de l'arbre}
  marquage(liste_position, A) {marquage de l'arbre}
fin si

3.2 Construction de l'arbre des suffixes

Nous construisons l'arbre des suffixes du texte analysé à partir de l'algorithme de Chen et Seiferas adapté pour la construction d'arbre des suffixes généralisés [Hui92]. Cet algorithme permet la construction de l'arbre en temps et en espace linéaire sur la taille du texte. On utilise pour cela des "liens raccourcis" permettant à chaque étape de trouver en temps constant la position d'insertion dans l'arbre d'un suffixe à partir de celle du suffixe précédent. Le texte est exploré caractère par caractère de la droite vers la gauche. Il y a au plus $|\Sigma|$ liens raccourcis associés à chaque noeud de l'arbre. Un noeud représentant le sous-mot w est relié par le lien raccourci d'étiquette $a \in \Sigma$ au noeud contenant le plus court sous-mot de préfixe aw. Les liens sont mis à jour et

rajouté après l'insertion de chaque nouveau noeud en temps constant. Nous devons toutefois adapter la construction pour rendre l'arbre utilisable pour l'algorithme de Cobbs de recherche de motifs avec erreurs. Pour cela, il est nécessaire d'associer à chaque noeud de l'arbre un attribut correspondant au nombre d'occurrences dans le texte du sous-mot associé au noeud. Cet attribut vaut donc 1 pour les feuilles et, pour tout noeud interne, la somme du nombre de feuille contenues dans le sous-arbre issu de ce noeud. Cela correspond à l'attribut Nb utilisé dans l'algorithme de Cobbs. On calcule la valeur de tous les attributs en temps linéaire sur le nombre de noeuds (et donc sur la taille du texte) par un simple parcours de l'arbre après sa construction.

L'algorithme de Cobbs nécessite aussi l'utilisation de liens suffixes différents des liens raccourcis nécessaires à la construction de l'arbre. Il existe un lien suffixe associé à chaque noeud. Pour un noeud représentant le sous-mot aw, avec $a \in \Sigma$, il y a un lien suffixe vers le noeud représentant le sous-mot w. Or avec l'algorithme de Chen et Seiferas, pour tout noeud aw il existe un lien raccourci vers le noeud w. On peut donc construire les liens suffixes en rajoutant les liens inverses aux liens raccourcis. Etant donné qu'il y a au plus autant de liens suffixes qu'il y a de noeuds dans l'arbre, on peut construire ces liens en temps linéaire sur le nombre de noeuds. Cette construction se fait au cours de l'étape de calcul de l'attribut Nb qui nécessite le parcours de tous les noeuds de l'arbre. Le temps de construction de l'arbre ainsi que son adaptation pour l'algorithme de Cobbs est donc linéaire sur la taille du texte.

3.3 Ordonnancement de la recherche des sous-motifs

De la même façon que dans l'algorithme de Myers, il est clair que l'ordre dans lequel sont recherchés les sous-motifs est primordial pour l'efficacité de la recherche. En effet, moins un sous-motif aura d'occurrences dans le texte, moins il y aura de positions possibles pour la recherche du sous-motif suivant et donc plus l'élagage de l'arbre sera efficace. De plus, moins un sous-motif représentera de sous-mots différents et plus la recherche dans l'arbre sera rapide. Il faut donc trouver un critère pour déterminer la potentialité d'un sous-motif en terme de nombre d'occurrences et de sous-mots représentés.

Contrairement à Myers qui propose de déterminer son critère de fréquence par un tirage aléatoire de texte et une recherche du nombre d'occurrences du sous-motif, nous voulons éviter d'avoir recours à une simulation coûteuse en temps et nous proposons une estimation analytique de notre critère.

Dans un premier temps, on peut remarquer que les deux critères qui nous intéressent sont liés : moins un sous-motif représentera de sous-mots différents, moins il aura d'occurrences dans le texte. Il faut donc pour tout sous-motif appartenant au langage qui nous intéresse être capable de dénombrer les sous-mots différents qu'il représente.

De façon à optimiser le temps de recherche au maximum, l'ordonnancement qui va nous intéresser est un *ordonnancement consécutif*. Etant donné un motif $M = M_0 S_1 M_1 S_2 M_2 \dots S_p M_p$, à chaque étape de la recherche, on a testé l'appariement d'une suite de sous-motifs M_k à M_h et on cherche le prochain appariement avec le sous-motif M_{k-1} ou le sous-motif M_{h+1}. On peut ainsi tirer parti de la position des sous-motifs déjà trouvés pour élaguer la recherche du sous-motif suivant dans l'arbre.

Pour des raisons de simplicité, nous ne tiendrons pas compte ici des erreurs de types insertion et suppression. Ce qui nous intéresse est moins de dénombrer exactement les sous-mots différents que de trouver un critère donnant un ordonnancement de la potentialité des sous-motifs en terme de nombre d'occurrences. Les insertions et suppressions n'étant prises en compte pour aucun des sous-motifs, on peut supposer que l'ordonnancement sera peu altéré. De plus, nous ne présentons ici que l'ordonnancement dans le cas d'une recherche de motif "linéaire" c'est à dire ne contenant pas d'union entre sous-motifs. Nous présentons toutefois dans la partie traitant de l'extension à l'union la méthode pour réaliser l'ordonnancement dans le cas d'un motif quelconque.

Dans un premier temps, on calcule le nombre de sous-mots différents représentés par un mot de longueur m, sur un alphabet Σ avec k erreurs (nous ne considérons pas l'ambiguïté sur les caractères). Le nombre de sous-mots différents possibles q doit être la somme des sous-mots contenant $k, k-1, \dots, 0$ erreurs. Pour un nombre d'erreur l donné, on doit choisir leurs l positions dans le sous-mot ce qui représente C_m^l choix et pour chaque combinaison possible il y a $|\Sigma| - 1$ façons possibles de remplacer le caractère du sous-motif par un caractère erroné. On obtient donc le calcul de q :

Formule 1

$$q = \sum_{l=0}^{k} C_m^l (|\Sigma| - 1)^l$$

Dans un deuxième temps, nous considérons la possibilité d'ambiguïté sur les caractères du sous-motif. La différence vient ici du fait qu'une erreur qui a lieu à

une position correspondant à un caractère ambigu ne donne plus $|\Sigma| - 1$ façons de remplacer le caractère. On doit tenir compte du nombre de caractères représentés par le caractère ambigu.

Soit σ_i le nombre de caractères non représentés par le caractère ambigu i. On a alors σ_i caractères correspondant à une erreur. Si on appelle classe de caractère ambigu un ensemble de caractères ambigus qui représente le même nombre de caractères de l'alphabet Σ, on a $|\Sigma|$ classes de caractères ambigus possibles.

Soit f le nombre de classes de caractères ambigus apparaissant dans le sous-motif, on doit faire maintenant une répartition des k erreurs parmi ces f classes.

Comme dans notre précédent calcul de q (formule 1), on commence par choisir le nombre l d'erreurs, puis on choisit $l_1 \leq min(l, m_1)$ avec m_1 le nombre de positions dans le sous-motif où apparaît un caractère ambigu de classe 1 et l_1 le nombre d'erreurs effectivement à l'une de ces positions. De la même façon on définit $l_2 \leq min(l - l_1, m_2)$ le nombre d'erreurs restantes se trouvant à une position correspondant à un caractère ambigu de classe 2 et ainsi de suite jusqu'à $l_f \leq min(l - l_1 - \ldots - l_{f-1}, m_f)$. Si on pose $L_i = l - \sum_{j=1}^{i-1} l_j$, on obtient ainsi la nouvelle formule de calcul de q :

Formule 2

$$q = \sum_{l=0}^{k} \sum_{l_1=0}^{min(L_1,m_1)} \sum_{l_2=0}^{min(L_2,m_2)} \cdots \sum_{l_f=0}^{min(L_f,m_f)} \prod_{i=1}^{f} C_{m_i}^{l_i} (\sigma_i)^{L_1}$$

Grâce à ce nombre , nous allons pouvoir déterminer la formule de complexité de recherche d'un sous-motif en fonction des sous-motifs précédemment trouvés. Nous utiliserons ensuite cette formule pour déterminer un ordonnancement optimal des sous-motifs à rechercher.

La complexité temporelle de la recherche d'un sous-motif dans le texte, de l'extension de ces positions à droite et du marquage de l'arbre est en $O(|\Sigma|mq + N(\Delta + 2k) \log_{|\Sigma|} n + 2kmN)$ avec N son nombre d'occurrences. Si on note h le hième sous-motif cherché, on en déduit une formule qui permet d'estimer le nombre d'opérations à effectuer pour la recherche d'un sous-motif en fonction des précédents sous-motifs.

Soit q_h le nombre de sous-mots dans l'arbre qui vont correspondre au sous-motif. La formule 2 permet de calculer la valeur de q_h en fonction du nombre et de la classe des caractères ambigus contenus dans h ainsi que du nombre d'erreurs autorisées sur h.

Si on note f_h la fréquence supposée d'apparition d'un sous-motif à une position donnée, on a :

Formule 3

$$f_h = \frac{nombre\ de\ combinaisons\ du\ sous-motif}{nombre\ de\ mots\ possibles\ de\ longueur\ m_h} = \frac{q_h}{|\Sigma|^{m_h}}$$

Sur un intervalle de longueur Δ_h la probabilité d'apparition du sous-motif h est donc de $\Delta_h f_h$. Soit N_h le nombre d'occurrences supposées du sous-motif h seul, on a :

Formule 4

$$N_h = f_h(n - m_h)$$

On en déduit le nombre d'occurrences supposées dans un intervalle de distance d'un sous-motif précédent $N'_h = N_h(\Delta_{h-1} f_{h-1})$ et donc son nombre d'occurrences supposées en fonction des h-1 sous-motifs précédents :

Formule 5

$$N'_h = N_h(\prod_{i=0}^{h-1} \Delta_i f_i)$$

De la même façon, q'_h le nombre de sous-mots de l'arbre se détermine par :

Formule 6

$$q'_h = q_h(\prod_{i=0}^{h-1} \Delta_i f_i)$$

En fonction du sens s (0 ou 1) de la recherche du sous-motif h (à gauche ou à droite du sous-motif précédent), on peut estimer en fonction de q'_het de N'_h le temps T_h^s nécessaire pour sa recherche :

Formule 7

$$T_h^s = |\Sigma| m_h q'_h + N'_h(\Delta_{h+s} + 2k_h) \log_{|\Sigma|} n + 2k_h m_h N'_h$$

On peut à partir de la formule 7 en déduire une formule récursive donnant la meilleure extension possible à partir de deux sous-motifs g et d donnés (g et d sont les bornes gauche et droite de la recherche courante) :

Formule 8

$$Meilleur(g,d) = (min(T_g^0 + Meilleur(g-1,d), T_d^1 + Meilleur(g,d+1)))$$
$$et\ Meilleur(0,p) = 0$$

Grâce à cette formule, on peut déterminer le meilleur ordonnancement par programmation dynamique pour un sous-motif de départ h en calculant $|\Sigma|m_h q_h + Meilleur(h,h)$. On en déduit le meilleur ordonnancement quel que soit le sous-motif de départ en calculant $|\Sigma|m_h q_h + Meilleur(h,h)$ pour tous les sous-motifs h du motif P cherché. La complexité totale de ce calcul ne dépend que de la taille de la matrice de programmation dynamique utilisée et est donc en $O(|P|^2)$.

3.4 L'algorithme de Cobbs

3.4.1 L'algorithme de programmation dynamique

L'algorithme de programmation dynamique cherchant les appariements approximatifs entre S et P nécessite l'utilisation d'une matrice D à m+1 lignes et n+1 colonnes. On pose D(i,0)=0 et on construit par récurrence les autres entrées D(i,j) par la formule :

$$D(i,j) = min \begin{cases} D(i-1,j) + d(P_i, \epsilon) \\ D(i-1,j-1) + d(P_i, T_j) \\ D(i,j-1) + d(\epsilon, T_j) \end{cases}$$

D(i,j) est alors le coût minimum de tout appariement entre $P_{1...i}$ et un suffixe de $T_{1...j}$. L'ensemble des appariements approximatifs entre S et P est alors l'ensemble des j tels que $D(m,j) \leq k_{max}$.

Il n'est pas nécessaire de construire la matrice en entier. En effet, si l'on appelle essentielle une entrée (i,j) de D telle que $D(i,j) \leq k_{max}$, il est clair que toutes les entrées calculées à partir d'entrées non essentielles n'ont pas besoin d'être calculées car leur valeur ne peut être que supérieure de par la formule de calcul de D(i,j).

Si l'on prend par exemple $S = atacatacatcat$, $P = agacatgc$ et $k_{max} = 2$, la partie essentielle de la matrice D correspond à la figure 3. Le nombre associé à chaque case est D(i,j) et à partir de chaque (i,j) est représenté le chemin jusqu'à la première ligne de D correspondant à l'alignement de coût minimal entre $P_{1...i}$ et un suffixe de $T_{1...j}$. Dans la dernière ligne du tableau, aux colonnes 8 et 11, on trouve les deux occurrences de l'appariement approximatif entre P et S.

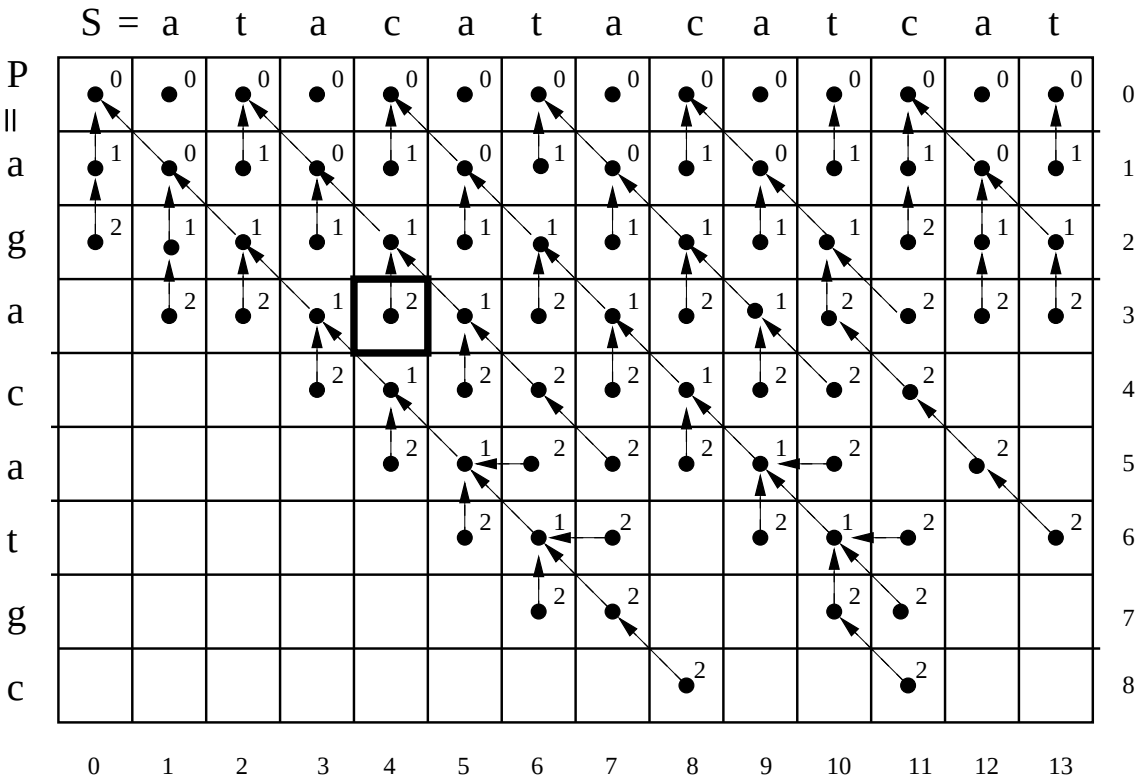


FIG. 3 – *Matrice de programmation dynamique pour l'alignement entre P=agacatgc et T=atacatacatcat avec un maximum de deux erreurs*

3.4.2 Modifications apportées à l'algorithme de programmation dynamique

L'algorithme de Cobbs est basé sur deux optimisations fortes de l'algorithme de programmation dynamique.

La première vient de l'utilisation de l'arbre des suffixes pour décomposer la recherche en deux parties, une partie où l'on recherche les appariements de mots possibles et une partie où l'on recherche l'occurrence de ces mots dans S. En effet, on s'aperçoit que l'algorithme de programmation dynamique réalise un grand nombre de calculs inutiles: pour toutes les répétitions contenues dans le texte on vérifie l'appariement avec le sous-motif alors qu'il suffirait de le faire pour une seule occurrence de chaque répétition. Grâce à l'arbre des suffixes, on peut dans une pre-

mière étape vérifier l'appariement avec tous les sous-mots différents du texte. Dans une deuxième étape, pour tous les sous-mots retenus comme s'appariant avec le sous-motif, on peut utiliser l'arbre pour retrouver (en temps linéaire en fonction du nombre d'occurrences) la position de toutes les occurrences dans le texte de ces sous-mots. En repoussant ainsi le plus tard possible le moment où l'on s'occupe des positions effectives dans le texte, on évite son parcours, ce qui permet d'obtenir un algorithme indépendant de sa longueur.

La deuxième optimisation est liée à la structure de la matrice de programmation dynamique et au fait que pour une entrée donnée il existe plusieurs chemins possibles de même coût. Cela correspond à plusieurs appariements possibles se terminant à la même position j (du fait des erreurs de type insertion et suppression) entre le sous-motif et un sous-mot du texte. L'idée est ici de ne chercher que l'ensemble des positions finales du texte où il y a un appariement et non pas tous les appariements différents possibles. On choisit de ne s'intéresser qu'aux plus courts appariements à une position donnée de la matrice. Cela sera suffisant car on ne perd pas de position d'appariement en ne conservant que les plus courts à une position donnée. En effet, tous les appariements plus longs ont comme suffixe le sous-mot le plus court et donc pour chacune de leurs occurrences, le sous-mot le plus court est aussi présent. On évite ainsi une trop forte combinatoire dans l'ensemble des sous-mots de l'arbre pouvant s'apparier pour ne conserver que ceux permettant de localiser l'ensemble des positions du texte où un appariement à lieu.

A chaque étape de l'algorithme, on cherche à trouver les noeuds de l'arbre d'une longueur donnée susceptibles de s'apparier avec une partie du texte. La première optimisation utilise l'arbre pour sélectionner les noeuds candidats possibles à cette étape puis la deuxième optimisation, par l'intermédiaire de deux critères (test suffixe et test cosuffixe), permet d'élaguer les candidats. Les sous-mots ainsi sélectionnés à la fin d'une étape sont appelés *sous-mots viables.*

Notion de sous-mots viables

Il existe plusieurs chemins de même coût dans la matrice pour une entrée donnée. On appelle *chemin canonique* le chemin "le plus à droite". Ces chemins sont représentés dans la matrice de la figure 3. Cela revient à privilégier les transitions verticales puis diagonales puis horizontales.

Définition 3: Si $D(i,j) \leq kmax$ alors le suffixe de $S_{1...j}$ décrit par le chemin canonique issu de (i,j) est appelé le sous-mot viable pour l'entrée (i,j) et se note

$V(i,j)$. Il s'agit du suffixe le plus court de $S_{1...j}$ s'appariant avec $P_{1...i}$.

Par exemple, pour $D(3,4)$, il existe plusieurs sous-mots dont la distance d'édition avec $P_{1...3}$ est 2 : $d(aga, ac) = 2$ et $d(aga, atac) = 2$. Cependant, seul ac se trouve sur le chemin canonique, donc $V(3,4) = ac$

Définition 4 : Pour $0 \leq i \leq m$, $V_i = \{V(i,j)/1 \leq j \leq n\}$ est l'ensemble des sous-mots viables de la ligne i, c'est à dire s'appariant avec $P_{1...i}$.

Par exemple, dans la figure 3 on a :

$$
\begin{aligned}
V_0 &= \{\epsilon\},\\
V_1 &= \{\epsilon, a\},\\
V_2 &= \{\epsilon, a, ac, at\},\\
V_3 &= \{a, at, ata, ac, aca, atc\},\\
V_4 &= \{ata, atac, aca, acat, atc\},\\
V_5 &= \{atac, ataca, atacat, acata, atca\},\\
V_6 &= \{ataca, atacat, atacata, atacatc, atcat\},\\
V_7 &= \{atacat, atacata, atacatc\},\\
V_8 &= \{atacatac, atacatc\}.
\end{aligned}
$$

V_8 représente donc l'ensemble des sous-mots de S les plus courts s'appariant avec au plus kmax erreurs avec P pour toutes les positions de S.

Du fait que l'on ne s'intéresse qu'aux entrées essentielles, si $D(i,j) > kmax$ alors $V(i,j)$ n'existe pas. On définit alors $Vcumul(i,j) = V(i',j)$ tel que $i' \leq i$ est la dernière ligne de la matrice telle que $V(i',j)$ existe. Ainsi, $Vcumul(i,j)$ existe pour tout i et j. $Vcumul_i$ se déduit facilement de $Vcumul_{i-1}$ et de V_i car $Vcumul(i,j) = V(i,j)$ si $V(i,j)$ existe et $Vcumul(i,j) = Vcumul(i-1,j)$ sinon.

Structure de V_i

L'algorithme de Cobbs est basé sur une construction ligne par ligne, construisant V_i à partir de V_{i-1}. On pose $V_0 = \{\epsilon\}$. Tout élément de V_i peut être construit soit par extension verticale ou diagonale d'un élément de V_{i-1}, soit par extension horizontale d'un élément de V_i. Si wa est élément de V_i, on dit qu'il est vertical si $wa \in V_{i-1}$, diagonale si $w \in V_{i-1}$ et horizontal si $w \in V_i$. On en déduit la liste des candidats pour V_i :

- Tous les w tels que $w \in V_{i-1}$

- Tous les wa tels que $w \in V_{i-1}$ et il existe une branche a partant du noeud w de l'arbre des suffixes.

- Tous les wa tels que $w \in V_i$ et il existe une branche a partant du noeud w de l'arbre des suffixes.

Seuls sont conservés les sous-mots w tels que $d(P_{1...i}, w) \leq kmax$. Ceci permet de construire la liste des appariements possibles selon le critère d'optimisation grâce à l'utilisation de l'arbre.

Il faut ensuite définir un critère permettant de sélectionner les sous-mots viables (les plus courts) parmi les candidats de façon à obtenir la deuxième optimisation. On définit pour cela deux critères : le critère suffixe et le critère co-suffixe.

Critère suffixe :

Soit w un candidat. Soit w' un suffixe de w tel que $D_i(w') \leq D_i(w)$ alors w ne peut pas être viable. En effet, w' se retrouve dans tous les occurrences de w donc w ne peut pas être sur le chemin canonique car un chemin plus court et/ou de moindre coût représentant w' existe toujours. Cela revient à dire que si un suffixe de w a un coût inférieur ou égal à w alors w n'est pas viable.

Par exemple, considérons l'entrée $(3, 4)$ de la figure 3. Les sous-mots atac et ac ont la même distance d'édition égale à 2 du préfixe du sous-motif aga. ac étant un suffixe de atac, atac ne peut être viable.

Critère co-suffixe :

Soit w un candidat et soit $v_1, v_2, \ldots v_r \in V_i$ tel que tout $v_s (1 \leq s \leq r)$ est un co-suffixe de w. D'après le critère précédent, si v_s est élément de V_i alors $D_i(v_s) < D_i(w)$. Donc, si à une position j où w apparaît, v_s est aussi présent alors $V(i, j) \neq w$. On en déduit que si pour toutes les occurrences de w il existe un co-suffixe de w dans V_i alors w ne peut être viable. C'est en quelque sorte le critère inverse du précédent.

Par exemple, dans la ligne 4 de la figure 3, aucun suffixe de ac n'a de score inférieur ou égal à $d(aga, ac) = 2$, donc ac n'est pas rejeté par le critère suffixe. Par contre, toute occurrence de ac est suffixe de atac de score $d(agac, atac) = 1$. Donc ac est rejeté par le critère co-suffixe.

3.4.3 Description de l'algorithme

L'algorithme comporte m itérations. Chaque itération i construit la liste V_i en étendant verticalement et diagonalement les éléments de la liste V_{i-1} et en étendant

horizontalement les éléments de la partie de la liste V_i déjà construite. On ne génère que les candidats w pour lesquels la distance d'édition est au plus kmax. Pour chaque candidat généré, on applique les deux critères de viabilité. Si w est viable il est rajouté à la liste V_i.

Génération des candidats

La liste V_i se construisant en partie à partir d'elle même. il est important de bien ordonner la génération des candidats de façon à être sûr qu'aucun candidat ultérieur ne sera susceptible de modifier leur validité par l'un des deux critères. Pour cela, les candidats sont générés par ordre croissant du nombre d'erreurs et pour un nombre d'erreurs donné, par ordre de taille croissante. Les listes V_{i-1} et V_i sont en fait constituées de kmax+1 listes chaînées, chacune contenant les sous-mots ayant même distance d'édition. Ces sous-listes sont elles même triées par ordre croissant de taille des sous-mots.

Les candidats de la sous-liste de score k sont générés à partir des sous-mots des sous-listes $k - d(P_i, \epsilon)$ de V_i pour l'extension verticale, $k - d(P_i, a)(a \in \Sigma)$ de V_{i-1} par l'extension diagonale et de $k - d(\epsilon, a)(a \in \Sigma)$ de V_i par l'extension horizontale. Chaque sous-liste k de V_i nécessite donc le parcours de $2|\Sigma| + 1$ sous-listes. Pour un score et une longueur fixés, les candidats sont générés par extension verticale d'abord puis par extension diagonale puis par extension horizontale.

Test du critère suffixe

Soit w un nouveau candidat sélectionné. Tout d'abord, on peut faire une première constatation : il est inutile de tester le critère suffixe pour un candidat généré par une extension verticale. En effet, si w était éliminé par le critère suffixe à l'étape i, cela voudrait dire que w était éliminé par la critère suffixe à l'étape i-1, donc w n'aurait pas été viable à l'étape i-1 et donc w ne pourrait pas être candidat à l'étape i. Seuls les candidats générés par extension diagonale et horizontale nécessitent d'être testés.

Pour déterminer si wa est éliminé par le critère suffixe, il suffit de trouver un de ses suffixes w'a déjà présents dans V_i (conséquence de l'ordonnancement par score/longueur des listes V_{i-1} et V_i). Pour cela, on parcourt les liens suffixes à partir du noeud wa jusqu'à ce que l'on trouve un $w'a \in V_i$, auquel cas wa est éliminé, ou jusqu'à ce que l'on atteigne la racine de l'arbre et que wa soit donc conservé comme candidat.

Test du critère co-suffixe

Ce test nécessite l'utilisation d'un compteur de co-suffixe associé aux noeuds de

l'arbre. Soit w un noeud et J l'ensemble des positions finales dans S où w apparaît. On a alors $|J| = Nb(w)$. Avec ces notations, on déduit que w est éliminé par le critère co-suffixe si pour toutes les occurrences $j \in J$, $V(i,j)$ est un co-suffixe de w. Si l'on note $Cos_i(w)$ le compteur co-suffixe associé à w, $Cos_i(w)$ dénombre pour combien de $j \in J$, $V(i,j)$ est un co-suffixe de w. On peut voir $Cos_i(w)$ comme la part des occurrences de w "couvertes" par un co-suffixe meilleur. Ainsi, on pourra éliminer w par le critère co-suffixe lorsque $Cos_i(w)$ sera égal à $Nb(w)$.

Nous allons voir maintenant comment ces compteurs sont mis à jour de façon à ce que, quand un candidat est généré, le compteur qui lui est associé soit à la bonne valeur.

Au début de l'itération i on met à jour les compteurs par $Cos_i(w) = Cos_{i-1}(w)$. Ensuite, toute augmentation du compteur $Cos_i(w)$ ne peut venir que de l'ajout d'un co-suffixe w' de w tel que $w' \in V_i$ et $w' \notin Vcumul_{i-1}$. Cela vient du fait que le seul moyen pour que la couverture de w augmente est qu'un nouveau sous-mot de V_i viennent couvrir des occurrences de w qui ne l'étaient pas encore.

Supposons maintenant que l'on vienne juste d'ajouter w' à V_i et que $w' \notin Vcumul_{i-1}$. Il y a donc des suffixes w de w' dont les compteurs co-suffixes doivent être mis à jour. Soit $h = Nb(w') - Cos_i(w')$, h est strictement positif car on a vu que quand $Nb(w') = Cos_i(w')$ alors w' n'est plus viable. h représente donc la partie des occurrences de w' qui ne sont pas couvertes. On ajoute h à tous les compteurs co-suffixes des suffixes de w' (en effet, la partie couverte de w' à déjà été ajoutée précédemment aux compteurs co-suffixes des suffixes de w') jusqu'au premier suffixe w de w' inclus, tel que w est viable. Il est suffisant de s'arrêter à w car tous les noeuds suffixes de w ont déjà été mis à jour (w est élément de V_i) et toutes les occurrences de w' sont aussi des occurrences de w.

On utilise aussi les compteurs co-suffixes pour mettre à jour $Vcumul_i$ par la propriété :

$w \in Vcumul_i$ ssi $w \in V_i$ ou $w \in Vcumul_{i-1}$ et à la fin de l'itération i, $Cos_i(w) < Nb(w)$.

Cobbs propose une optimisation de l'utilisation des compteurs co-suffixes de façon à faire un élagage plus rapide des candidats. L'idée est d'utiliser non plus un compteur co-suffixe associé à un noeud mais $|\Sigma|$ compteurs, $Cos_i^a(w)$, un pour chaque extension wa de w. $Cos_i^a(w)$ représente le nombre d'occurrences $j \in J$ pour lesquelles un co-suffixe w' apparaissant aussi en j est élément de $Vcumul_i$ et que

$S_{j+1} = a$. C'est en fait une partition du précédent compteur en $|\Sigma|$ compteurs et on a :

$$Cos_i(w) = \sum_{a \in \Sigma} Cos_i^a(w)$$

Pour mettre à jour les compteurs, il suffit cette fois de propager $|\Sigma|$ valeurs au lieu d'une seule. Il y a deux avantages à cette utilisation. Premièrement, il devient inutile d'étendre $w \in V_i$ horizontalement à wa si $Cos_i^a(w) = Nb(wa)$, chaque extension étant couverte par l'extension w'a de w' qui couvre $Cos_i^a(w)$.

De la même façon, il est inutile d'étendre $w \in V_i$ diagonalement à wa si $Cos_{i-1}^a(w) = Nb(wa)$.

Algorithme de Cobbs

Nous donnons maintenant une reconstitution rationnelle de l'algorithme complet de Cobbs ainsi que les principales fonctions utilisées à partir de la présentation qu'il en fait dans sa thèse [Cob95a].

algorithme 3 Procédure Cobbs

```
pour i = 1 à m faire
  pour tout k ≤ kmax faire
    pour tout l < i + k faire
      pour tout P ∈ V_{i-1}(k − d(P_i, ε)) et |P| = l faire
        {extension verticale}
        test co-suffix(P)
      fin pour
      pour tout a ∈ Σ faire
        pour tout P ∈ V_{i-1}(k − d(P_i, a)) et |P| = l faire
          {extension diagonale}
          Su ⇐ suffix(P)
          si non(test suffixe(Su)) alors
            si Cos^a_{i-1}(P) < Nb(Pa) alors
              si non(test co − suffixe(Pa)) alors
                {ajoute P à la liste V_i et à la liste Vcumul_i}
                ajout(P, V_i), ajout(P, Vcumul_i)
              fin si
            fin si
          fin si
        fin pour
      fin pour
      pour tout a ∈ Σ faire
        pour tout P ∈ V_i(k − d(ε, a)) et |P| = l faire
          {extension horizontale}
          Su ⇐ suffix(Pa)
          si non(test suffixe(Su)) alors
            si Cos^a_i(P) < Nb(Pa) alors
              si non(test co − suffixe(Pa)) alors
                {ajoute P à la liste V_i et à la liste Vcumul_i}
                ajout(P, V_i), ajout(P, Vcumul_i)
              fin si
            fin si
          fin si
        fin pour
      fin pour
    fin pour
  fin pour
  Mise_a_jour(Vcumul_i)
  efface(Vcumul_{i-1}), efface(V_{i-1})
fin pour
```

algorithme 4 Procédure testsuffixe(N)

{Si N est différent de la racine de l'arbre}
si $N \neq racine$ **alors**
  si $N \in V_i$ **alors**
    return VRAI
  sinon
    {on suit le lien suffixe à partir de N pour obtenir le noeud S}
    $S \Leftarrow suffixe(N)$
    testsuffixe(S)
  fin si
sinon
  return FAUX
fin si

algorithme 5 Procédure testco-suffixe(P)

$Cos_i(P) = \sum_{a \in \Sigma} Cos_i^a(P)$
si $Cos_i(P) = Nb(P)$ **alors**
  return VRAI
sinon
  si $P \notin Vcumul_{i-1}$ **alors**
    pour tout $a \in \Sigma$ **faire**
      $h[a] \Leftarrow Nb(Pa) - Cos_i^a(P)$
      {on suit le lien suffixe à partir de N pour obtenir le noeud S}
      $S \Leftarrow suffixe(P)$
      testco-suffixe2(S,h)
    fin pour
  fin si
  return FAUX
fin si

algorithme 6 Procédure testco-suffixe2(N)

{Si N est différent de la racine de l'arbre}
si $N \neq racine$ **alors**
  pour tout $a \in \Sigma$ **faire**
    $Cos_i^a(N) \Leftarrow Cos_i^a(N) + h[a]$
  fin pour
  si $N \notin V_i$ et $N \notin Vcumul_{i-1}$ **alors**
    {on suit le lien suffixe à partir de N pour obtenir le noeud S}
    $S \Leftarrow suffixe(N)$
    testco-suffixe2(S,h)
  fin si
fin si

algorithme 7 Procédure Mise_a_jour($Vcumul_i$)

pour tout $q \in Vcumul_{i-1}$ **faire**
  si $Cos_i(q) < Nb(q)$ **alors**
    {ajoute q à la liste $Vcumul_i$}
    $ajout(q, Vcumul_i)$
  fin si
fin pour

3.4.4 Analyse de la complexité

Par souci d'efficacité, les sous-listes V_i et $Vcumul_i$ sont représentées sous forme de tables de hachage.

Soit $q = |Vcumul_i|$. A chaque itération i, les candidats sont générés par extension des sous-mots viables de V_{i-1} et de Vi. Chaque w de V_{i-1} est étendu verticalement à au plus un candidat et diagonalement à au plus $|\Sigma|$ candidats. Chaque w de V_i est étendu horizontalement à au plus $|\Sigma|$ candidats. Donc $O(|\Sigma|(|V_{i-1}|+|V_i|))$ candidats sont générés à l'itération i.

$Vcumul_i = \bigcup_{j \leq i} V_j$, donc il y a $O(|\Sigma|q)$ candidats générés au cours de l'algorithme. Pour chacun d'eux, on applique les tests des critères suffixes et co-suffixes. Or au cours de ces tests on parcourt les liens suffixes jusqu'à ce que l'on trouve un élément de V_i ou de $Vcumul_i$ ou que l'on atteigne la racine de l'arbre. Il y a au plus $|w|$ suffixes à tout candidat w, donc au plus m tests de présence dans les listes V_i et $Vcumul_i$.

Dans la dernière étape de l'algorithme, pour tout élément de V_m on parcourt le sous-arbre issu du noeud correspondant pour trouver toutes les occurrences dans S de ces éléments. Ce parcours est linéaire en fonction du nombre d'occurrences.

Au total l'algorithme a donc une complexité temporelle en $O(|\Sigma|mq + nO)$.

Cobbs [Cob95a] a mené un certain nombre d'expérimentations sur la taille des listes $Vcumul_i$ et V_i en fonction de n, m et kmax. La taille de $Vcumul_i$ étant majorée par $\sum_{j \leq i} V_j$, c'est le comportement des listes V_i qu'il convient d'étudier plus précisément. Il a montré que la taille des listes V_i est fortement croissante tant que i est inférieur à $\log_\Sigma n + kmax$. Puis le nombre de sous-mots différents de longueur i existant dans le texte devenant de plus en plus réduit, la taille de V_i diminue fortement. Pour un nombre d'erreurs donné, la taille de m importe donc peu pour déterminer la taille des listes. Cela rend l'algorithme particulièrement efficace dans le cas où les sous-motifs sont longs et ne contiennent pas trop d'erreurs. Il donne un exemple dans lequel $n \simeq 3$ millions de caractères, $m = 30$ et $kmax = 6$. Dans ce cas la taille maximale de $Vcumul_i$ est $|Vcumul_i| \simeq 360000$ ce qui reste donc inférieur à n. Cobbs n'a toutefois pas fait l'étude de l'évolution de la taille des listes V_i en fonction de la taille des séquences. On ne connaît pas le facteur d'accélération de la recherche dans le cas de séquences plus longues (de l'ordre du milliards de caractères) correspondant à l'analyse d'une banque de séquences ou au génome humain complet par exemple.

3.5 Recherche d'un sous-motif

La recherche d'un sous-motif se fait par l'algorithme de Cobbs grâce à un parcours de l'arbre des suffixes. Une première étape nous donne la liste des sous-mots différents les plus courts s'appariant avec le sous-motif. Une deuxième étape nous permet de trouver, par un simple parcours de l'arbre à partir des sous-mots repérés, l'ensemble des positions du texte correspondant au début d'une occurrence d'un des sous-mots. Sachant la longueur des sous-mots, nous obtenons la liste de toutes les positions finales des occurrences de tous les sous-mots s'appariant avec le sous-motif. Dans le cas de la recherche à droite du prochain sous-motif, cette liste est suffisante car on recherche la position du sous-mot suivant à partir de la position finale du sous-mot courant. Par contre, dans le cas de la recherche à gauche, nous allons devoir (comme dans l'algorithme de Myers) chercher les extensions possibles à gauche des sous-mots trouvés étant donné que pour une position finale donnée nous n'avons repéré que le plus court sous-mot s'appariant avec le sous-motif. En étendant à gauche tous les sous-mots trouvés nous allons trouver tous les débuts possibles des sous-mots s'appariant avec le sous-motif. Or, pour un sous-motif donné, la longueur d'un sous-mot s'appariant avec lui ne peut varier qu'en fonction du nombre d'erreurs autorisées. Nous sommes assuré par l'algorithme de Cobbs d'avoir trouvé l'occurrence la plus courte à une position donnée. Donc l'extension maximale vers la gauche du sous-mot ne peut représenter que $k+m-l$ caractères avec m la longueur du sous-motif et l la longueur du sous-mot considéré. En effet, le plus long sous-mot pouvant s'apparier avec un sous-motif de longueur m est de longueur $m+k$ dans le cas où il y a k erreurs de type insertions. La plus longue extension possible à gauche d'un sous-mot de longueur l ne peut donc se faire que sur $k+m-l$ caractères.

Pour chaque extension possible, il suffit de vérifier s'il existe un appariement avec au plus k erreurs entre le sous-mot étendu et le sous-motif. Donc pour chaque occurrence dans le texte du sous-motif, on fait une extension en $O(km)$ pour obtenir la liste de tous leurs débuts.

3.6 Structure de la liste de positions

Après avoir ordonnancé la recherche des sous-motifs, on procède à la recherche des occurrences du premier sous-motif. Le sous-motif suivant est recherché soit à gauche du premier, c'est à dire dans un intervalle de positions déterminées par rapport au début de celui-ci, soit à droite du premier et donc dans un intervalle de positions déterminées par sa fin. On obtient alors un nouveau sous-mot (appelé groupe de sous-mots) correspondant à l'association des deux premiers sous-mots et

possédant ses propres début et fin. Ce groupe de sous-mots correspond à un appariement de l'association des deux sous-motifs (appelés groupe de sous-motifs) avec le texte. Chaque étape suivante consiste en l'agrégation de nouveaux sous-mots au groupe de sous-mots déjà construit dans l'ordre déterminé par l'ordonnancement et en la mise à jour des nouveaux débuts et fins correspondant. L'algorithme s'arrête après l'ajout du dernier sous-motif permettant d'obtenir la liste de ces occurrences. En cas de recherche à gauche du groupe de sous-motif courant, l'extension à gauche de tous les groupes de sous-mots correspondants permet de trouver l'ensemble des nouveaux groupes de sous-mots (voir figure 4). Lors de la recherche à droite, par contre, l'extension à gauche du nouveau sous-mot est inutile car seule sa fin nous intéresse (voir figure 5). L'algorithme de Cobbs assurant de trouver toutes les positions de fin des sous-mots s'appariant, on est sûr de ne perdre aucune occurrence.

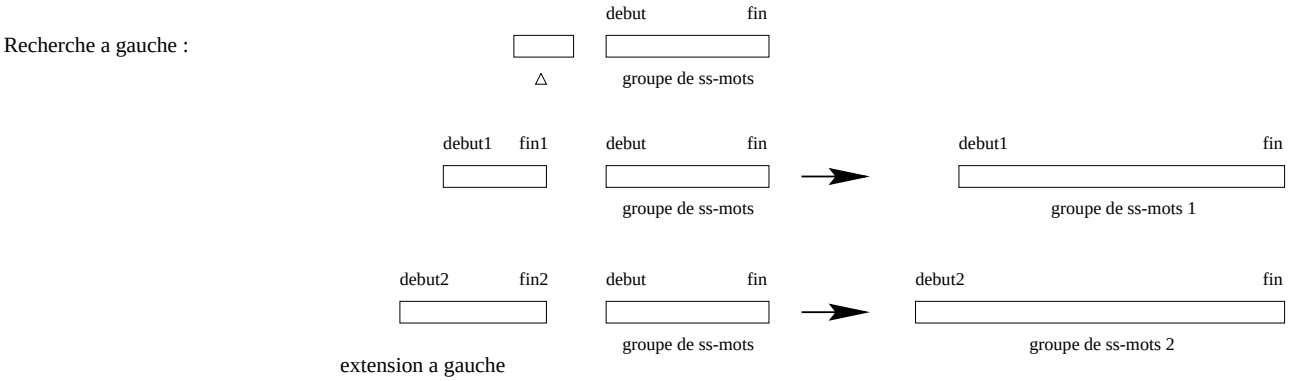


FIG. 4 – *Recherche à gauche du prochain sous-motif et effet de l'extension à gauche*

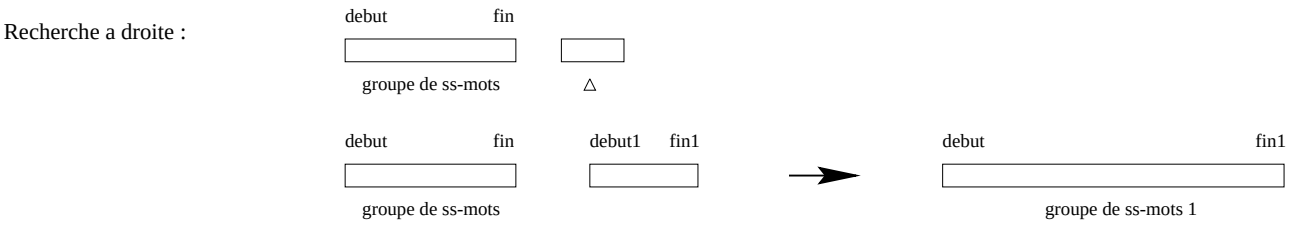


FIG. 5 – *Recherche à droite du prochain sous-motif*

Un groupe de sous-mots correspond donc à l'association de plusieurs sous-mots séparés par des intervalles de distances et est entièrement caractérisé par son début

et sa fin. La liste des positions conservées entre chaque étape de l'algorithme est donc une liste de groupes de sous-mots. Chaque élément de cette liste est constitué de deux pointeurs sur le texte, l'un pour le début du groupe de sous-mots et l'autre pour sa fin.

3.7 Marquage dans l'arbre des débuts possibles de motifs

Etant donné un groupe de sous-mots, on effectue un marquage de l'arbre de façon à optimiser la recherche du sous-motif suivant en fonction de l'intervalle de distances autorisées. Pour cela, on veut pouvoir retrouver dans l'arbre (en temps linéaire sur le nombre de groupes de sous-mots considérés) l'ensemble des feuilles correspondant à un suffixe du texte dont le début est situé dans un intervalle de distances autorisées de l'un des groupes de sous-mots.

Nous utilisons une structure de type vecteur de pointeurs (appelée vecteur de référence) permettant à partir d'une position quelconque du texte de retrouver instantanément la feuille correspondant au suffixe commençant à cette position. Cette structure est construite en même temps que l'arbre. Au début de la construction de l'arbre elle est initialisée comme un vecteur de n pointeurs à nil. A chaque nouvelle feuille créé, nous rajoutons un pointeur à partir de la position du vecteur correspondant au début du suffixe considéré vers cette feuille (voir figure 6). Ceci ce fait en temps constant pour chaque feuille et ne modifie donc pas la complexité temporelle de la construction de l'arbre des suffixes.

L'arbre des suffixes étant une structure non symétrique, elle ne permet de retrouver directement que les positions des débuts des sous-mots contenus dans le texte. Le marquage de l'arbre doit donc se faire différemment dans le cas de la recherche à gauche et de la recherche à droite.

Dans le cas de la recherche à gauche, on a la position du début du groupe de sous-mots courant et on veut chercher des sous-mots s'appariant avec le sous-motif suivant et dont la fin se trouve dans l'intervalle de distances autorisées. Or dans l'arbre des suffixes, ce sont les débuts des sous-mots qui sont représentés dans les feuilles. Pour déterminer l'ensemble des feuilles représentant des suffixes dont le préfixe est susceptible de se trouver dans le bon intervalle de distance Δ, il faut considérer la longueur maximale que pourra avoir un sous-mot s'appariant avec le sous-motif. Sachant que l'on autorise k erreurs dans cet appariement, le plus long sous-mot aura pour longueur $m + k$ (m étant la longueur du sous-motif) et le plus court $m - k$. Si on a $\Delta = [\Delta_i, \Delta_s]$, alors l'intervalle des débuts de sous-mots possibles est $[\Delta_i + m - k, \Delta_s + m + k]$ (voir figure 7) de façon à garantir de n'omettre aucune

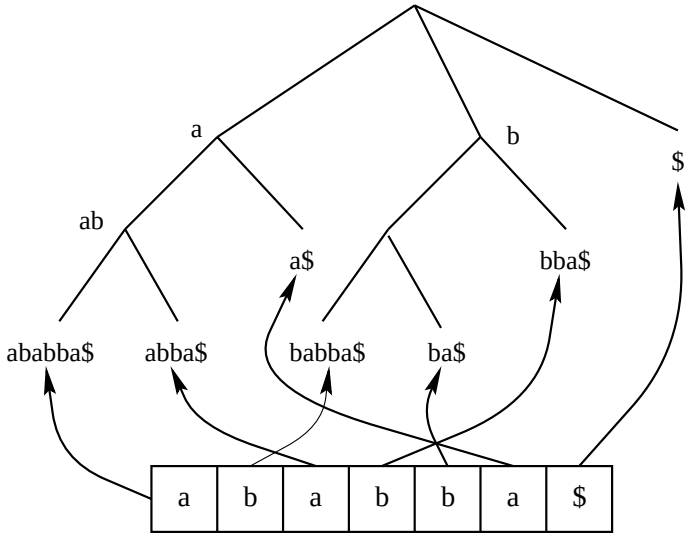


FIG. 6 – *Utilisation du vecteur de référence pour l'arbre des suffixes de la chaîne "ababba$"*

occurrence.

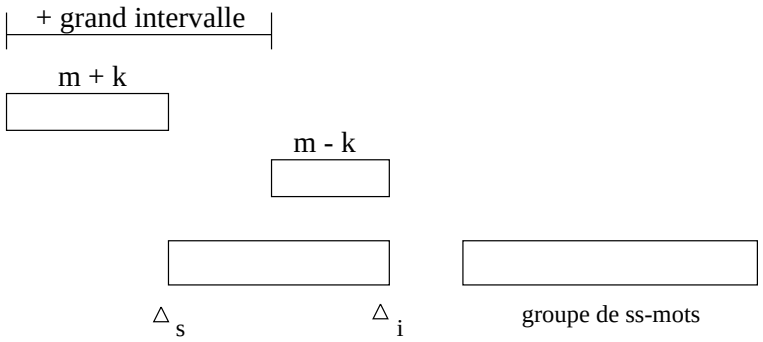


FIG. 7 – *Détermination du plus grand intervalle possible pour le début du sous-mot dans le cas de la recherche à gauche*

On ne garantit cependant pas avec un intervalle aussi large que toutes les occurrences trouvées soient valides (par exemple un sous-mot de longueur $m-k$ dont le début est à la distance $m+k+\Delta_s$). Il faut donc lors de la phase de récupération

des occurrences dans l'arbre vérifier que la fin de chaque sous-mot trouvé se situe bien dans l'intervalle autorisé.

Dans le cas de la recherche à droite, on dispose cette fois de la position de fin du groupe de sous-mots courants et on veut les sous-mots dont le début se trouve dans l'intervalle de distances autorisées. Il suffit donc de conserver l'intervalle tel quel : $\Delta = [\Delta_i, \Delta_s]$ car l'arbre nous permet d'avoir directement accès aux positions de début des sous-mots.

Une fois déterminé l'ensemble des positions valides pour les occurrences du sous-motif suivant, on utilise le vecteur de référence pour retrouver les feuilles correspondantes. Puis, à partir de chacune de ces feuilles, on remonte dans l'arbre en suivant les liens pères en marquant chaque noeud rencontré et cela jusqu'à arriver sur la racine ou sur un noeud précédemment marqué. Il est en effet inutile de continuer après avoir rencontré un noeud déjà marqué car cela signifie que lors d'une étape précédente il a été marqué ainsi que tous les noeuds situés entre lui et la racine.

Il faut maintenant modifier l'algorithme de Cobbs pour tenir compte du marquage de l'arbre. Si un noeud a été marqué cela signifie qu'il existe parmi les feuilles du sous-arbre issu de ce noeud au moins une feuille correspondant à un suffixe du texte susceptible d'avoir pour préfixe une occurrence du sous-motif dans un intervalle autorisé. Dans le cas contraire, il n'y a aucune feuille dans le sous-arbre répondant à ce critère, il est donc inutile de continuer la recherche à partir de ce noeud. Le sous-mot correspondant est donc éliminé des candidats à la viabilité. Seuls les noeuds marqués sont donc conservés dans la liste des candidats.

Cette méthode permet un élagage d'autant plus important des candidats que le nombre de noeuds marqués est faible, c'est à dire que le nombre d'occurrences du groupe de sous-mots précédents est peu élévé. Donc, l'élagage devient de plus en plus efficace au fur et à mesure que l'on rajoute des sous-motifs à la recherche courante du fait de la réduction progressive du nombre d'appariements. C'est aussi ce qui fait l'efficacité de l'ordonnancement de la recherche en permettant de rechercher en premier les sous-motifs qui auront le moins d'occurrences et qui élagueront donc l'arbre le plus rapidement.

La complexité temporelle du marquage de l'arbre est liée aux nombre d'occurrences du groupe de sous-mots précédents, à la taille de l'intervalle autorisé ainsi qu'a la profondeur moyenne dans l'arbre des feuilles considérées. Pour chaque occurrence du groupe de sous-mots précédents et pour chacune des positions correspondant à l'intervalle de distances autorisées, on a, au plus, une remontée dans l'arbre d'un

nombre de noeuds égal à la profondeur de la feuille. Or l'arbre des suffixes a une profondeur moyenne de l'ordre de $\log_{|\Sigma|} n$. La complexité totale de ce marquage est donc en $O(N \log_{|\Sigma|} n\Delta)$ avec N le nombre d'occurrences du groupe de sous-mots précédents et Δ la taille de l'intervalle autorisé.

3.8 Remise à zéro de l'arbre

Un problème se pose toutefois avec cette méthode de marquage. Lors de la recherche d'un sous-motif, seuls les noeuds liés à l'intervalle, considéré à cette étape, doivent être marqués. Les noeuds qui ont été marqués lors des étapes précédentes doivent donc être "remis à zéro" dès que l'on a trouvé toutes les occurrences qui leur correspondent. Or une remise à zéro qui nécessiterait un parcours de l'arbre aurait une complexité temporelle linéaire en fonction de la taille du texte, ce que l'on veut absolument éviter. Il faut donc trouver une méthode pour réaliser cette remise à zéro en un temps indépendant de la taille du texte.

Nous proposons ici une méthode permettant de réaliser la remise à zéro en temps constant. L'idée est de considérer les marqueurs comme des pointeurs vers un objet de référence pouvant prendre une valeur binaire. Au début de l'algorithme tous les noeuds de l'arbre pointent vers un objet de référence de valeur 0. Dès que l'on veut chercher un nouveau sous-motif, on crée un objet de référence de valeur 1. Lors de la phase de marquage, on déplace simplement le pointeur de l'objet de valeur 0 à l'objet de valeur 1. A la fin de la phase de marquage, tous les noeuds marqués pointent sur l'objet de valeur 1 et les autres vers un objet de valeur 0. La remise à zéro se fait alors instantanément par l'affectation de la valeur 0 à l'objet de valeur 1. De nouveau, tous les noeuds de l'arbre pointent vers un objet de valeur 0 (voir figure 8). Il suffit de répéter ce principe pour chaque étape de l'algorithme en créant à chaque fois un nouvel objet de valeur 1.

3.9 Motifs non linéaires

Nous avons considéré jusqu'à présent uniquement des motifs linéaires c'est à dire des motifs ne contenant pas d'union entre sous-motifs et où les seules unions possibles ont lieu au niveau des caractères (caractères ambigus). Nous présentons ici une extension permettant de prendre en compte l'utilisation de l'union entre sous-motifs ou groupes de sous-motifs.

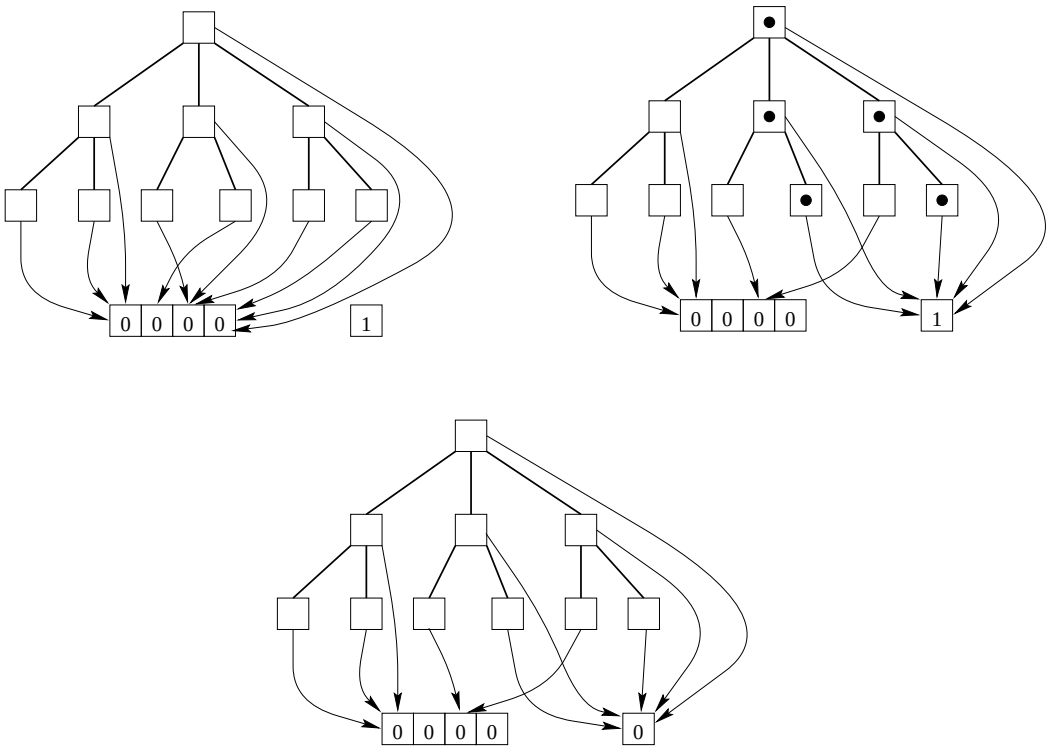


FIG. 8 – *Exemple de remise à zéro de l'arbre en temps constant*

Un groupe de sous-mots est représenté par une liste de positions. L'union entre deux groupes de sous-mots correspond à l'union des positions des occurrences de chacun des deux groupes. L'union de deux groupes de sous-mots peut donc se faire par la fusion des deux listes de positions les représentant.

Les unions de groupes de sous-motifs pouvant être imbriquées (un groupe de sous-motifs pouvant lui aussi être composé d'unions de groupes de sous-motifs), il faut organiser la recherche du motif complet récursivement. A chaque étape, si le groupe de sous-motifs considéré est composé d'une union de sous-motifs, on éclate la recherche sur chacun des membres de l'union et on concatène les listes de sous-mots résultats à la fin de chaque recherche (voir algorithme 2). Il faut donc modifier le calcul de l'ordonancement pour tenir compte de ce mécanisme. La formule de calcul de l'ordonancement devient le formule 9.

Formule 9

$$Meilleur(g,d) = (min(\sum_{g_i \in g}(Meilleur(g_i,d)) + Meilleur(g-1,d), \sum_{d_i \in d}(Meilleur(g,d_i)) + Meilleur(g,d+1)))$$

$$Avec \sum_{g_i \in g} Meilleur(g_i,d) = T_g^0 si\ g\ contient\ un\ seul\ g_i$$

$$Et \sum_{d_i \in d} Meilleur(g,d_i) = T_d^1 si\ d\ contient\ un\ seul\ d_i$$

Dans cette formule, on décompose le sous-motif (où groupe de sous-motifs) g (resp. d) courant en ses différentes composantes d'union g_i (resp. d_i) et on fait la somme de leurs valeurs à laquelle on ajoute la meilleure décomposition pour le sous-motif suivant.

4 Conclusion

Nous avons présenté dans ce papier un outil particulier de recherche de motifs couplés avec erreurs dans des textes basés sur l'arbre des suffixes. Il se décompose en deux parties permettant une optimisation à deux niveaux de la recherche de tels motifs. La première correspond à l'utilisation de l'algorithme de Cobbs pour accélérer l'algorithme de programmation dynamique permettant la recherche d'un motif unique avec erreurs. La deuxième est liée à la recherche de sous-motifs couplés. Nous proposons une méthode basé sur un ordonnancement des motifs cherchés par un critère analytique donnant la potentialité d'un sous-motif en terme d'appariement avec une séquence aléatoire ainsi que sur un élagage de l'arbre des suffixes pour optimiser la recherche des sous-motifs successifs. Grâce à ces deux approches complémentaires notre outil permet une recherche de motifs complexes en un temps indépendant de la taille du texte analysé.

Nous avons développé FOREST, un environnement d'analyse lexicale pour les grandes séquences génétiques (plusieurs millions de caractères) [GN96]. Cet outil est basé sur deux structures de données complémentaires permettant une grande variété d'opérations pour le visualisation de la structure lexicale de ces séquences. La première structure est un arbre dictionnaire des répétitions exactes qui est un arbre des suffixes adapté pour l'analyse des séquences biologiques. La deuxième structure

est un vecteur booléen représentant l'ensemble des positions dans la séquence.

Notre outil de recherche de motifs complexes va être associé à FOREST de façon à fournir un environnement complet de visualisation et d'analyse lexicale des séquences biologiques. L'originalité de notre outil vient du fait qu'il permet une recherche multiple de motifs complexes en un temps indépendant de la taille du texte analysé. Il est donc particulièrement bien adapté pour la recherche de motifs dans les grandes séquences biologiques dont la longueur est en constante augmentation.

Nous allons valider notre outil en l'utilisant pour l'analyse de génomes complets comme *Escherichia coli* [KVD95] représentant près de 5 millions de caractères. Nous étudierons plus spécifiquement la présence associé d'un motif particulier (le motif de *Shine-Dalgarno*) avec le début des gènes.

Références

[AP83] A. Apostolico and F.P. Preparata. Optimal off-line detection of repetitions in a string. In *Theorical Computer Science*, volume 22, pages 297–315, Holland, 1983.

[Apo85] A. Apostolico. The myriad virtues of subword trees. In *Combinatorial algorithms on words*, pages 85–96, 1985.

[Bai92] A. Bairoch. A dictionary of sites and patterns in proteins. *Nucleic Acids Research*, 20:2013–2018, 1992.

[BBH+85] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Comput. Sci.*, 40:31–55, 1985.

[BJEG95] A. Brazma, I. Jonassen, I. Eidhammer, and D. Gilbert. Approaches to the automatic discovery of patterns in biosequences. *Reports in informatics University of Bergen*, 113, 1995.

[BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. In *Commun. ACM*, volume 20, pages 762–772, 1977.

[BYG92] R. Baeza-Yates and G.H. Gonnet. A new approach to text searching. In *Commun. ACM*, volume 35(10), pages 74–82, 1992.

[Cob95a] A. L. Cobbs. *Approximating the Suffix Tree.* Computer science, University of California at Berkeley, 1995.

[Cob95b] A.L. Cobbs. Fast approximate matching using suffix trees. In *Combinatorial Pattern Matching 1995*, volume 937, 1995.

[Cro88] M. Crochemore. String matching with constraints. *Mathematical Fondation of Computer Science*, 3:44–58, 1988.

[CS85] M.T. Chen and J. Seiferas. Efficient and elegant subword-tree construction. In *Combinatorial algorithms on words*, pages 97–107, 1985.

[EM96] N. El-Mabrouk. Recherche approchée de motifs. application à des séquences biologiques structurées. In *PhD Thesis*, University of Paris VII, 1996.

[EMC96] N. El-Mabrouk and M. Crochemore. Boyer-moore strategy to efficient approximate string matching. *Lecture notes in computer science*, 1075:24–38, 1996.

[GN96] R. Gras and J. Nicolas. Forest, a browser for huge dna sequences. In *The Seventh Workshop on Genome Informatics 96*, Tokyo, December 1996.

[GP90] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. on Computing*, 19:989–999, 1990.

[Hui92] L.C.K. Hui. Color set size problem with application to string matching. In *Combinatorial Pattern Matching*, pages 230–243, Springer-Verlag, 1992.

[JCH95] I. Jonassen, J.F. Collins, and D.G. Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Science*, 4(8):1587–1595, 1995.

[JE95] I. Jonassen and I. Eidhammer. Discovering patterns conserved in sets of related protein sequences. In *Proceedings of Norwegian Informatics Conference*, pages 95–112, Norway, 1995.

[KMGL87] S. Karlin, M. Morris, G. Ghandour, and M.Y. Leung. Efficient algorithms for molecular sequence analysis. In *Proc. Natl. Acad. Sci. USA*, volume 85, pages 841–845, 1987.

[KMP77] D.E. Knuth, J.H. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323–350, 1977.

[KVD95] F. Kunst, A. Vassarotti, and A. Danchin. Organization of the european bacillus subtilis genome sequencing project. *microbiology*, 141:249–255, February 1995.

[LV88a] G.M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoret. Comput. Sci.*, 43:239–249, 1988.

[LV88b] G.M. Landau and U. Vishkin. Fast string matching with k differences. *J. Comp. Sys. Sci.*, 37:63–78, 1988.

[McC76] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the association for computing machinery*, 23(2):262–272, 1976.

[MHEM84] M.E. Mulligan, D.K. Hawley, R. Entriken, and W.R. McClure. Escherichia coli promoter sequences predict in vitro rna polymerase selectivity. *Nucleic Acids Research*, 12:789–800, 1984.

[MM93] G. Mehldau and E.W. Myers. A system for pattern matching applications on biosequences. *Cabios*, 9(3):299–314, 1993.

[Mye94] E.W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12:345–374, 1994.

[Mye96] E.W. Myers. Approximate matching of network expression with spacers. *Journal of Computational Biology*, pages 33–51, 1996.

[PBPR89] J. Postfai, A.S. Bhagwat, G. Postfai, and R.J. Roberts. Predictive motifs derived from cytosine methyltransferases. *Nucleic Acids Research*, 17:2421–2435, 1989.

[TU93] J. Tarhio and E. Ukkonen. Approximate boyer-moore string matching. *SIAM J. Comput.*, 22:243–260, April 1993.

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6:132–137, 1985.

[Ukk93] E. Ukkonen. Approximate string-matching over suffix trees. In *Combinatorial Pattern Matching 1993*, volume 4, pages 228–242, 1993.

[Wei73] P. Weiner. Linear pattern matching algorithms. In IEEE Computer Society, editor, *14th Annual symposium on switching and automata theory*, pages 1–11, 1973.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399